The Fun-Project: From Requirements Specification to Program Presentation

Michal Iglewski, Marcin Kubica, Jan Madey, Janina Mincer-Daszkiewicz, Krzysztof Stencel

ABSTRACT

This paper describes a software engineering project which evolved from certain critical control applications. Its goal is to improve a professional practice in the design and implementation of safe, reliable computer systems. This is to be achieved by development of a proper methodology together with supporting software tools. Since this methodology is addressed to practitioners, it must be not "too academic" but at the same time theoretically sound.

1 Introduction

Software systems are undoubtedly among the most complex artifacts made by humankind but also among the least trustworthy. These two facts are clearly related. Errors in software are not caused by a fundamental lack of our knowledge. In principle, we know everything about the effect of each instruction that is executed. Software errors are caused by our inability to fully understand all the interrelations within these complex products.

Many software systems are offered to the market though their producers know that these products are not error-free. Such a situation is not acceptable, especially in safety-critical applications — systems must work when needed. The purpose of the project being described in this paper is to investigate the whole process of software construction ("from requirements specification to program presentation") and to suggest a possible approach which may result in a well documented, good quality software.

This paper is an extended version of [12]. In particular, in appendices we present several examples in order to illustrate better the described methods. The research presented here was originated by early works on "information hiding" by D.L. Parnas [20, 21], an experience resulted from the Software Cost Reduction project (A-7E) at the Naval Research Laboratory [3, 4, 7, 8, 19], recent involvement of Parnas in investigating of the shutdown software at the Ontario Hydro's Darlington Nuclear Power Generating Station [23, 27], and by our cooperation with Parnas, primarily in years 1990-91 [5, 13, 24, 25].

1.1 The A-7E experience

The purpose of the A-7E project was to redesign and rebuild the operational flight program for the Navy's A-7 aircraft in order to evaluate the applicability of new software engineering techniques for embedded real-time systems. It gave a chance to practically verify a number of ideas developed by scientists, and has resulted in the extended literature and some follow-up works. The A-7E project was conducted by the Naval Research Laboratory and the Naval Weapons Center, Washington, DC in the late 70th and early 80th, but international Software Cost Reduction workshops (having as a main goal reporting new work in extending the A-7 model) are still being organized, and our current research is strongly influenced by the early SCR work at NRL.

The project is too large and complex to be even briefly described here but let us at least mention a few observations resulting from the A-7E experience, and having an impact on our work presented in the rest of this paper.

- An adequate communication and specification language between specialists in different areas is crucial (pilots versus computer specialists, in this case). It must be precise, yet understandable. Hence, (a) only simple, traditional mathematics is preferable, (b) tabular notation is very useful.

- A decomposition of a complex problem into smaller subproblems is indispensable. A proper modularization is very important, and the "black box" approach seems to be the good one.
- In safety critical control applications a mere testing of software is not sufficient and a form of formal verification must be also required.
- A proper documentation in software engineering is critical.

1.2 The Darlington experience

Current Canadian design practice in the nuclear industry requires three systems that are capable of shutting down the reactor in the case of an incident. The main reactivity control system (which controls the production of power) monitors the performance of the plant and can also cause power generation to stop if anything goes wrong. In addition, there are two safety (*shutdown*) systems; their only task is to shut the system down if anything abnormal occurs. During normal operation the two shutdown systems have no function other than collecting and displaying data to be used in determining if the plant is functioning as it should.

In earlier Canadian reactors the shutdown systems were not computerised but constructed of analogue devices and relays. They were simple, easily studied, and trustworthy. However, this technology requires a lot of equipment and also could not perform sophisticated checking procedures. At the Darlington Nuclear Power Generation Station in Ontario, the two shutdown systems were, for the first time, implemented in software. This software was ready much earlier than the rest of the plant, had been tested thoroughly, and was considered by its owner — Ontario Hydro — to be safe to use. The Atomic Energy Control Board of Canada, however, was not willing to give the licence to operate the plant until they were convinced of the correctness of the shutdown software. Though the code itself was not too long, it was quite complex, and the requirements documentation was neither complete nor precise enough. As a result, the manufacturer was asked to produce a mathematical requirements document using [8] as a model. This document was reviewed by nuclear safety experts. It was also agreed that precise program documentation would be produced and used as the basis for an inspection procedure.

That was an expensive and painful process[1], involving some 60 people for several months. The lessons learned from the Darlington experience confirmed most of the A-7E observations, inspired our research described in the following sections, and in particular led to the development of the Display Method (cf. Section 5).

2 Fun paradigm

The project we are currently involved in (called the *Fun-Project*, from the term "functional approach" [25]) is based on the premise that professional documentation for the development of computer systems containing complex software can consist of representations of certain mathematical functions and relations, often presented in a tabular notation. In the following sections some basic ideas about the functional approach to documentation of computer systems are briefly introduced.

2.1 Documentation in computer system design

The starting point for a system design should be a description of the environment in which the system is to be used and a description of the system's required behavior. In our approach the system is modelled as a dynamic one, i.e., as a state machine that monitors and controls certain aspects of its environment[2]. Those aspects determine the *environmental quantities of interest*. There may be physical constraints on those quantities, and the computer system will be required to further restrict them. All this information is recorded in the *system requirements document*. Next, certain decisions are to be made, and recorded in the *system design document*, about hardware that will be used. This

1. Some details about it can be found e.g. in [23, 27].
2. The state of the system is a total function of time.

document describes the interfaces to the I/O devices by presenting the relations between the I/O registers and the environmental quantities. The software designer's job is to implement a mapping from computer inputs to computer outputs such that the required relation between the environmental quantities of interest is satisfied. The system requirements and system design documents can be combined into a single one, called the *software requirements document*. In such an approach a design of a computer system begins with a "black-box" description of the system as a whole, a "clear-box" description of the hardware architecture, and a "black-box" description of the software (cf. [5, 7, 8, 24, 25]).

Non-trivial software systems are designed and constructed by teams, each representing certain knowledge and experience. Hence, an important task is to subdivide the whole construction job into several smaller work assignments. Each assignment is to design — and later on to implement — a *module*, consisting of an internal (*private, hidden*) data structure and a group of programs. Those programs of the module which can be used from outside this module form an *abstract interface* of this module and are called the *module's access-programs*; they are the only means to access the internal data structure of the module. Our approach to specification of module interfaces (the *trace assertion method*; in short: *TAM*) is briefly discussed in Section 3. The structure of the software system, presented in a *software module guide* [4], should indicate the design decisions and describe "secrets" of each module.

For every implementation of a module there should be a document describing the *module internal design*, i.e., the internal data structures and the effect of the module's access-programs on the state of that structure (cf. Section 4). Programs within modules should be presented and documented in a way allowing their inspection and maintenance. For this purpose we propose the *display method* (cf. Section 5).

2.2 Functional approach — four relations model

The contents of all mentioned above documents are defined in [24, 25]. These papers contain a more general discussion of the role and the structure of documentation in software engineering. Below we present some intuition behind it and explain certain basic concepts. In particular, the four relations: NAT, REQ, IN, and OUT are informally introduced.

2.2.1 The system requirements document

The values of environmental quantities of interest (those to be measured and/or controlled) change with time. Hence, each of them can be expressed as a *time-function* in the way that is usual in engineering (i.e., by modelling time as a set of real numbers). The association between the physical quantities of interest and their mathematical representations must be carefully defined. Having done that, we may now express in terms of binary relations:

- constraints on the values of environmental quantities placed by nature and previously installed systems (the relation called *NAT*), and
- further constraints on the values of environmental quantities, which the system under construction is required to impose (the relation called *REQ*).

The domain of these relations consists of *monitored state functions*, $\boldsymbol{m} = (m_1, \ldots, m_q)$, where each m_i corresponds to the i-th monitored quantity, while the range consists of *controlled state functions*, $\boldsymbol{c} = (c_1, \ldots, c_p)$, where each c_i corresponds to the i-th controlled quantity. If the same quantity is to be both monitored and controlled, this fact must be specified by NAT.

The requirements should specify behavior for all cases that can arise, but at the same time should not demand the impossible. Hence, we introduce a notion of *feasibility of REQ with respect to NAT*: nature allows at least the required behavior; it does not mean, however, that the functions involved are computable or that a practical implementation is possible.

2.2.2 The system design document

The next step is to identify the computers within the computer system and describe how they communicate, with emphasis on the peripheral devices. Hence, new sets of quantities are distinguished – *input* and *output registers* – which are also represented by time-functions. The additional environmental quantities can be classified as follows:

- *inputs*, quantities that are read by the computers in the system, associated with input registers on those computers, and represented by a tuple $\boldsymbol{i} = (i_1, \ldots, i_u)$;
- *outputs*, quantities whose values are set by the computers in the system, associated with output registers on those computers, and represented by a tuple $\boldsymbol{o} = (o_1, \ldots, o_v)$.

The physical interpretation of the inputs and their connection with the monitored quantities should be specified by a relation called *IN*, whereas the effects of the outputs and their connection with the controlled quantities are specified by a relation called *OUT*.

2.2.3 The software requirements document

This *document* can be seen as a combination of the two previously described documents, and hence it would contain the four relations: NAT, REQ (feasible with respect to NAT), IN, and OUT. The software implementation will yield a system with input-output behavior that can be described by yet another relation, *SOF*. It means, in particular, that:

- domain(SOF) contains all physically possible instances of $\boldsymbol{i}$, i.e., domain(SOF) $\supseteq$ range(IN);
- range(SOF) contains all possible instances of $\boldsymbol{o}$, i.e., range(SOF) $\subseteq$ domain(OUT).

SOF will be a function if the software is to be deterministic.

The resulting software must satisfy all the requirements expressed by the relations. Hence, the following should hold:

$$(\mathrm{NAT} \cap (\mathrm{IN} \bullet \mathrm{SOF} \bullet \mathrm{OUT})) \subseteq \mathrm{REQ} \qquad (*)$$

If the relations REQ, IN, OUT, and SOF are functions, we can use functional notation to rewrite (*) as follows:

$$\forall \boldsymbol{m} \; [\; (\boldsymbol{m} \in \mathrm{domain(NAT)}) \Rightarrow (\mathrm{REQ}(\boldsymbol{m}) = \mathrm{OUT}(\mathrm{SOF}(\mathrm{IN}(\boldsymbol{m})))) \;] \qquad (**)$$

The writers of the requirements document must deliver NAT, REQ, IN, and OUT, whereas the implementors determine SOF and verify (*) or (**). A document of this type will require natural language in the description of the environmental quantities, but can otherwise be precise and mathematical. The use of natural language in the definition of the physical interpretation of mathematical variables is unavoidable and quite usual in engineering.

3 The trace assertion method for module interface specification

As already mentioned, software should be hierarchically structured and consist of a collection of information-hiding modules, each one of them introduces a new type of *objects* and forms an *abstract data type. A* module interface specification gives a complete “black-box” description of the behavior of the module’s objects allowing designers of other modules to do their work without having any knowledge about the internal structure of the module. Hence only *externally observable* features of objects introduced by a module should be described on this level of abstraction.

The *trace assertion method* (in short: TAM) is a formal method for abstract specification of interfaces of modules which fulfills the above stated requirements. The method was first presented by Bartussek and Parnas [1], and then investigated and modified a number of times (e.g. [26]). A revised and updated version of TAM is under preparation [11]; prototype software tools supporting its practical application are already in use [10].

3.1 Traces

An *object* in TAM behaves like a finite state machine (a Mealy machine)[3], i.e., it has *states*, produces *outputs,* and can be affected by *events*. An external observer can perceive only events affecting the object and the outputs produced in response to these events. A (*feasible*) *trace* is a finite sequence of pairs (*E,O*), where *E* is an event, and *O* is the respective output. In the case of deterministic behavior of the object (to which case we will limit our discussion in this paper) we do not need to include outputs in traces and hence a trace is written as follows:

$$E_1 \bullet E_2 \bullet \ldots \bullet E_n$$

The dot is also used as an operator defined on traces. If T_1 and T_2 are traces then $T_1 \bullet T_2$ is a trace obtained by concatenation of T_1 and T_2. The empty sequence is called the *empty trace* and denoted by "_" (the underscore).

From the "black-box" point of view all traces after which the future behavior of the object is the same, are *observationally equivalent.* They can be grouped together and represented by a single *canonical trace.* Since we adopted a finite state machine model, the number of possible *equivalence classes* is finite and hence the number of canonical traces is also finite. A canonical trace represents the abstract value of the object.

3.2 Trace specifications

A module implements homogenous and independent objects. Hence, we may assume that there is only one, *generic*, object in a module. It communicates with other modules by means of:

- *access-programs* — a set of programs that can be used by objects from other modules to provide information to, and/or receive information from the object;
- *input variables* — a vector of external variables that the object observes;
- *output variables* — a vector of variables whose values are computed by the object and can be observed externally.

Thus, the following events can affect an object:

- *access-program invocations* — calls of access-programs with actual arguments,
- *input variable events* — selected changes of values of the input variables;

and the following outputs can be produced by an object:

- *values returned* by access-program invocations via arguments and/or directly,
- *values of output variables*.

A trace specification consists of five sections discussed below (c.f. Appendix A and E).

The Characteristic Section

It lists specific features of the module (e.g. the name of the specified type, parameters, foreign types, if any).

The Syntax Section

It presents:

- a list of input and output variables, and a definition of input variable events,
- a list of access-programs with a description of arguments and outputs produced by each access-program.

The Canonical Trace Section

It contains a definition of the set of canonical traces (in terms of the characteristic predicate). If auxiliary functions are introduced to simplify the specification, their definitions should be written in this section as well.

---

3. A more detailed discussion on fundamentals of TAM is to be found in [14].

The Equivalence Section

It contains a definition of the trace equivalence relation by describing a set of *extension functions* that map from single event extensions of canonical traces to the equivalent canonical traces (and status tokens, giving additional information for the designers of the module). For each access-program we first specify the intended invocations by defining the "legality" function, returning a status token (%legal% for the intended case, %fatal% for invocations not guaranteeing termination). Next, we assume arguments leading to the token %legal% and only for such a domain we specify the value of the extension function, i.e., a new canonical trace. If the value of status token is different from %legal% and %fatal%, then we assume that the canonical trace stays the same.

The Return Value Section

It contains a definition of the output relation, i.e., a specification of returned values different than those being defined by the module. It is a relation, and not a function, since in a general case we allow non-determinism here. If the invocation is not legal (the token is different than %legal%), the returned values can be stated as *undefined*.

3.3 Miscellaneous

Writing non-trivial specifications is not an easy task. We need software tools supporting preparation of documentation, and in particular for checking the correctness of a specification against its syntax and semantics. Some work on such tools is briefly described in Section 7.

4 Internal design of modules specified by the trace assertion method

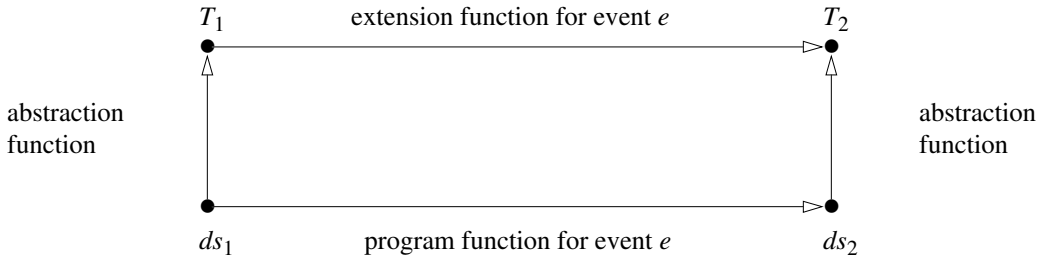
Before an implementation of a module begins, but after a programming language is decided, a "clear-box" specification of that module is to be written. It should reveal the module's "secret" by defining the internal data structure to be used for the representation of the abstract values, and the effects of events in terms of state changes of this data structure, described by the *program functions*. The mapping from *concrete states* of the internal data structure to *abstract states* of objects (canonical traces) is described by the *abstraction function*. The correctness of the internal design with respect to the interface specification must be verified.

Mathematical concepts and the terminology used here are adopted from early papers by Hoare [9] and Mills [17, 18]. Details about the internal design are presented in [15].

4.1 Preliminaries

Changes of the concrete states are expressed by program functions *(*as introduced by Mills).

An internal design of a module must be consistent with the interface specification of this module (c.f. Appendix C). In particular, the following diagram should commute [25] (where ds_1 and ds_2 denote data structure's states, and T_1 and T_2 denote canonical traces):



Commutation of the diagram constitutes a proof obligation for the given internal design.

4.2 The structure of the internal design document

A collection of facts and decisions common to all internal design documents for a given project must be grouped together and precede those documents; this collection is called Project Guide. It should contain, in particular:

- a choice of programming language (in the examples in Appendices: standard Pascal),
- the matching between built-in abstract types and concrete types of the chosen programming language (in our examples: <int> $\stackrel{df}{=}$ integer, <bool> $\stackrel{df}{=}$ boolean).

Each internal design document consists of sections described below (cf. Appendix D and F).

The Characteristic Section

It lists specific features of the module and is similar to the Characteristic Section in a trace specification.

The Data Structure Section

It contains several (often optional) subsections describing the data structure, its initial values and constraints.

The Abstraction Function Section

It defines an abstraction function, *af*.

The Program Function Section

It presents (usually in a tabular form) program functions for each access-program. In this and other sections we use the following notational convention: " ‘s " (to be read: "s *before*") denotes the value of the argument "s" before the invocation of an access-program, whereas " s’ " (to be read: "s *after*") denotes its value after the invocation[4].

5 The display method for precise documentation of programs

Even after the best decomposition of software into modules we may still end up with programs whose internal behavior can be difficult to understand and thus hard to review and/or to modify. We believe that programs should be documented precisely, systematically and readably, making their verification and maintenance feasible and relatively simple. One of possible ways of achieving this goal is to use the approach advocated by the Display Method. This method is an improved version of the technique used in the inspection of safety-critical shutdown software for the Darlington Nuclear Power Generation Station in Ontario, Canada [23].

In the following sections this method will be briefly explained and illustrated by a simple example (Appendix D). A complete discussion of the Display Method is to be found in [27].

5.1 The main idea

A successful program has more readers than writers. The Display Method is based on a very simple idea — to understand a program we should present it in small portions ("displays"), in such a way that each portion can be studied without looking at the others. At the same time, however, we have to ensure that they fit together to make the whole program correct.

A well-structured program can usually be written as a short text in which names of other programs (*subprograms*) may appear. These subprograms can also be short and can include the names of other subprograms. By a *display* we mean a concise document that consists of the following three parts:

- P1: a specification for the program presented in this display,
- P2: the program itself with possibly names of subprograms appearing in this text; we say that these subpro-

4. Note that ‘s and s’, as mathematical variables, could have been replaced by other symbols, but we would then have to establish an explicit correspondence between those symbols and the arguments of the access-program.

grams are *invoked* in this display,

- P3: specifications of all subprograms invoked in P2 that are not known[5].

Note that a name appearing in the program P2 may represent a procedure call (in which case it will usually be followed by actual parameters) but may also be treated as a macro call, to be replaced by a sequence of instructions. In either case, the construction of the resulting program by merging the P2 parts of all displays should be a simple operation that can be done automatically.

To avoid repetition of information in several displays, we introduce a separate document, a *lexicon*, which contains definitions of terms used in the program being documented. It will include definitions of any mathematical functions, programs constants, types, etc. that are used in more than one display.

Next, we say that:

- a *display is correct* if the program in P2 will satisfy the specification in P1, provided that the subprograms invoked in P2 satisfy the specifications given in P3,
- a *set of displays is complete* if, for each specification of a subprogram that is found in P3 of a display, there exists another display in which this specification is in P1,
- a *set of displays is correct* if (1) the set of displays is complete, and (2) all displays are correct.

A display can be supplemented by an additional part, P4, that contains a demonstration of its correctness. This could be either a description of the informal reasoning or a more formal argument. The existence of this additional section would make the reviewer's task simpler.

The Display Method can be used with any specification technique. In our present works we use a refinement of Mills' approach[6] [17], since we found it suitable for large programs. Mills does not include axiomatic descriptions of programming language statements among his basic definitions. Instead, he assumes that the programs, from which other programs are constructed, can be described by mathematical functions or relations. Since this assumption is valid for all programs, one can apply Mills' approach even when the component programs are quite long and complex. This allows the same method to be used for well-structured programs of any size, while many other methods do not deal with the problem of how to assemble small programs into large ones.

In documentation, the notation is very important; documents are to be read by experts from a variety of fields and should be easily understood. Our approach, as it was already mentioned, is based on the use of tables to describe mathematical functions, relations, and sets [22] in a more readable manner.

5.2 Miscellaneous

Originally, the Display Method was intended for program presentation. It soon became clear that we should use it while developing programs — documenting programs using this method can result in significant improvements of their quality. However, we believe that tool support is needed to make it practical for real applications.

6 Examples

The Appendices contain examples illustrating the described documents. The syntax of trace specifications is demonstrated in Appendix A. It contains the specification of a simple software module implementing limited stacks of integers. The specification is parameterized with the capacity of a stack. Some syntax aspects of the document are explained in comments. These comments are attached only for the purpose of this presentation and do not constitute part of the specification.

The internal design for the stack module is given in Appendix B. It is preceded by the Project Guide which deliv-

5. A *known* program is one that does not require a specification since its semantics is assumed to be known.
6. Although Mills is the best known proponent of this approach, similar ideas were independently discovered by many others.

ers information common for all modules within one project.

According to formal documentation methodology consistency of a trace specification and its internal design should be formally verified. The way this verification should proceed is outlined in Appendix C.

The Display Method is illustrated in Appendix D with a set of displays for a small program inverting a list of $n \geq 1$ integers, $a_1, \ldots, a_n$. Inversion is correct if for every $1 \leq i \leq n$, $a_i' = `a_{n-i+1}$. This problem is solved with a help of a stack (as introduced in Appendix A).

A more elaborated example of a trace specification is presented in the last two Appendices. It specifies a device interface module. The device is the Attitude Director Indicator — one of the displays in the cockpit of the A-7 aircraft. The example serves two purposes. First, since it is one of the device interface modules specified in the A-7E documentation [3], it shows changes in the semantics and syntax of the trace assertion method. Second, it demonstrates the use of input and output variables. Generally, the specification follows that from [6], which closely follows the original one. The significant alteration concerns the interpretation of output variables. In [3, 6] the specification does not differentiate between environmental output variables which have physical interpretation and output variables created solely for methodological reasons. In our approach output variables are understood as a means to communicate the outputs by hardware [16] and are not used for other purposes. The other difference concerns the semantics of input events. In our approach the changes of input variables values are atomic [2], whereas in [3, 6] an input event may simultaneously change values of a number of input variables. This aspect is not, however, observable in the example specification.

More advanced applications of TAM are presented in other papers on this method (e.g. [2]).

7 Concluding Remarks

New methods and their foundations are being developed "by definition" at universities and scientific institutions. If, however, a method is to be verified on industrial examples and needs advanced supporting software tools, the university environment is not so suited any more. Scientists, teachers, and students do a very good job when working on a new, exciting idea. But at the end, a student wants a degree, a scientist needs publications; they are not interested in "dirty" aspects of practical work like finishing some tedious implementation details, writing documentation, meeting dead-lines, etc. In the case of tasks requiring a team work, we face additionally tremendous management problems: the job is to be done by people who are not really paid for such a work and hence it is very difficult to execute it. In the extreme case students simply disappear without leaving any useful results of their activities.

Knowing all those difficulties, we are trying, however, to build a set of integrated tools for the methods being developed within the Fun-Project[7]. This set contains at present:

- Fun-Spec — an editor supporting the Trace Assertion Method,
- Fun-Inter — an editor supporting the Internal Design, and
- PDS ("Prototype Display System") — a system supporting the Display Method.

For the first two tools we decided to use the Synthesizer Generator [29] as a development platform, because it provides all necessary mechanisms and significantly reduces time and effort needed to develop an editor. There were also other factors which determined our choice (e.g., a friendly user interface, a support for structural and textual editing, a chance to express and check various semantic properties). The Synthesizer Generator creates a "ready-to-run" editor according to the grammar and certain rules prepared by the designer of the editor. We have used the prototype versions of Fun-Spec and Fun-Inter for testing and education; we are quite satisfied with the results. It became obvious, however, that we should extend these tools by various other functionalities, in particular to allow more advanced

7. The Fun-Project is conducted jointly at the Warsaw University and the Universitè du Quèbec à Hull. There are also tool projects at McMaster University, supervised by D.L. Parnas.

verification of the specification, including its animation [10, 28].

PDS is a different tool, since we need not just an editor but a system supporting both the designers and reviewers of programs, including management aids. After the initial experiments with the first version of PDS, we are at present developing PDS+, in which, in particular, a commercial data base system will be incorporated.

While designing and implementing all these tools, we sometimes have to clarify and modify certain syntactical and semantic aspects of the methods. While applying the methods, we tempt also to modify them, what implies of course also a need for modification of tools. All this makes the whole project quite challenging (and never ending!).

Acknowledgements

Works by Dave Parnas and our cooperation with him greatly influenced our research interests, the project presented in this paper, and the paper itself. We are very grateful to him. The Fun-Project requires a team work and we would like to express our gratitude to all colleagues in Warsaw, Hull, and Hamilton for their contribution.

This work was partly supported by the State Committee for Scientific Research in Poland (KBN, grant 8 S503 040 04), by the Natural Sciences and Engineering Research Council of Canada (NSERC), by the NATO Linkage grant (HTECH. LG. 941314), and by Digital Equipment's European External Research Programme (EERP PL-002).

References

1. Bartussek, W., Parnas, D.L., "Using Traces to Write Abstract Specifications for Software Modules", in *Proc. 2nd Conf. of European Cooperation in Informatics,* Springer-Verlag, LNCS 65, 1978, pp. 211-236; Reprinted in Gehani, N., McGettrick, A.D. (Eds.), *Software Specification Techniques,* AT&T Bell Telephone Laboratories, 1985, pp. 111-130.
2. Bojanowski, J., Iglewski, M., Madey, J., Obaid, A., "Functional Approach to Protocols Specification", in *Protocol Specification, Testing and Verification XIV*, Vuong, S.T., Chanson, S.T. (Eds.), Chapman & Hall, 1995, pp. 395-402.
3. Britton, K.H., Clements, P.C., Parnas, D.L., Weiss, D.M., "Interface Specifications for the SCR (A-7E) Extended Computer Module", U.S. Naval Research Laboratory, Washington D.C., *NRL Memorandum Report 5502*, 1984, p. 129.
4. Britton, K.H., Parnas, "A-7E Software Module Guide", U.S. Naval Research Laboratory, Washington D.C., *NRL Memorandum Rep. 4702*, 1981, p. 32.
5. Engel, M., Kubica, M., Madey, J., Parnas, D.L., Ravn, A.P., van Schouwen, A.J., "A Formal Approach to Computer Systems Requirements Documentation", in Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (Eds.), *Hybrid Systems*, Springer-Verlag, LNCS 736, 1993, pp. 452-474.
6. Erskine, N., "The usefulness of the trace assertion method for specifying device module interfaces", *CRL Report* No. 258, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1992.
7. Heninger, K.L., "Specifying Software Requirements for Complex Systems: New Techniques and their Application", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, January 1980, pp. 2-13.
8. Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E., "Software Requirements for the A-7E Aircraft", U.S. Naval Research Laboratory, Washington D.C., *NRL Memorandum Report 3876*, November 1978, p. 523.
9. Hoare, C.A.R., "Proof of Correctness of Data Representations", *Acta Informatica*, vol. 1, no. 19, 1972, February, pp. 271-281.
10. Iglewski, M., Kubica, M., Madey, J., "Editor for the Trace Assertion Method", in *Proc. 10th International Conference CAD/CAM, Robotics and Factories of the Future: CARs & FOF'94,* Zaremba, M. (Ed.), OCRI, Ottawa, Ontario, Canada, 1994, pp. 876-881.
11. Iglewski, M., Kubica, M., Madey, J., Mincer-Daszkiewicz, J., Stencel, K., "Report on the Trace Assertion Meth-

od 95", *in preparation.*

12. Iglewski, M., Madey, J., "Software Engineering Issues Emerged from Critical Control Applications", in *Proc. 2nd IFAC Workshop, Safety and Reliability in Emerging Control Technologies*, Daytona Beach, Florida, USA, 1995, pp. 21-36.
13. Iglewski, M., Madey, J., Parnas, D.L., Kelly P. C., "Documentation Paradigms", McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, *CRL Report* 270, July 1993, p. 45.
14. Iglewski, M., Madey, J., Stencel, K., "On Fundamentals of the Trace Assertion Method", Warsaw University, Institute of Informatics, Warsaw, Poland, *Technical Report TR 94-09 (198)*, 1994, p. 8.
15. Iglewski, M., Mincer-Daszkiewicz, J., "Implementing Modules Specified in the Trace Assertion Method", presented at International Conference *Formal Specifications: Foundations, Methods, Tools and Applications* (*FM-TA'95*), Konstancin-Jeziorna, Poland, May 29-31, 1995.
16. Iglewski, M., Mincer-Daszkiewicz, J., Stencel, K., "Case Study in Trace Specification of Non-deterministic Modules", in *Proc. of the CS&P'95 Workshop*, Warsaw, Poland, October 11-13, 1995.
17. Mills, H.D., "Function Semantics for Sequential Programs", in *Proc. IFIP Congress 1980*, North Holland, 1980, pp. 241-250.
18. Mills, H.D., Basili, V.R., Gannon, J.D., Hamlet, R.G., *Principles of Computer Programming: A Mathematical Approach*, Allyn and Bacon, 1987.
19. Parker, A., Britton, K.H., Parnas, D.L., Shore, J., "Abstract Interface Specifications for the A-7E Device Interface Module", U.S. Naval Research Laboratory, Washington D.C., *NRL Memorandum Report 4385*, 1980, p. 151.
20. Parnas, D.L., "Information Distributions Aspects of Design Methodology", in *Proc. IFIP Congress '71*, Booklet TA-3, 1971, pp. 26-30.
21. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol. 15, no. 12, Dec. 1972, pp. 1053-1058.
22. Parnas, D.L., "Tabular Representation of Relations", McMaster University CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, *CRL Report* 260, 1992, p. 17.
23. Parnas, D.L., Asmis, G.J.K., Madey J., "Assessment of Safety-Critical Software in Nuclear Power Plants", *Nuclear Safety*, vol. 32, no. 2, 1991, pp. 189-198.
24. Parnas, D.L., Madey, J., "Documentation of Real-Time Requirements", in Kavi, K.M. (Ed.) *Real-Time Systems. Abstraction, Languages and Design Methodologies*, IEEE Computer Society Press, 1992, pp. 48-56.
25. Parnas, D.L., Madey, J., "Functional Documents for Computer Systems", *Science of Computer Programming,* vol. 25, no. 1, Oct. 1995, pp. 41-61.
26. Parnas, D.L., Wang, Y., "The Trace Assertion Method of Module Interface Specification", Queen's, C&IS, Telecommunication Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, *Technical Report 89-261*, 1989, p. 37.
27. Parnas, D.L., Madey, J., Iglewski, M., "Precise Documentation of Well-Structured Programs", *IEEE Transactions on Software Engineering*, vol. 20, no. 12, Dec. 1994, pp. 948-976.
28. Stencel, K., "Refined Simulation Techniques for the Trace Assertion Method", Warsaw University, Institute of Informatics, Warsaw, Poland, *Technical Report TR 95-17 (217)*, 1995, p. 11.
29. T*he Synthesizer Generator Reference Manual*, 4th edition, GrammaTech, Inc., Ithaca, NY, 1993.

Appendix A

Integer Stack Module

Informal Introduction

The specified module implements *limited stacks of integers*. PUSH puts its argument on the top of the stack. POP removes the top element without returning its value. TOP returns the top integer without changing the stack at all.

(0) CHARACTERISTICS

• type specified: <stack>

• parameters: max: <int>

Comments:

Names of types are written in angle brackets. The parameter "max" specifies the capacity of stacks; <int> is a built-in integer type.

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Value Type
PUSH	<stack>:VO	<int>:V	
POP	<stack>:VO		
TOP	<stack>:V		<int>

Comments:

The descriptor "V" means that the value of the argument may be used by the program. The descriptor "O" means that a value will be returned in this argument.

(2) CANONICAL TRACES

canonical(T) $\Leftrightarrow \exists n$: <int>; $x_1,\ldots,x_n$: <int> $(T = [PUSH(*, x_i)]_{i=1}^{n} \wedge n \leq max)$

Comments:

- The abstract value of a stack can be represented as a possibly empty sequence of invocations of the access-program PUSH, with the first argument (the name of the stack) being irrelevant and hence replaced by the symbol "*". (We will be using "*" to replace an obvious or irrelevant information in other contexts, as well.)
- The notation "$[Q(i)]_{i=1}^{n}$" describes the trace Q(1).….Q(n), if n > 0, and the empty trace, if n ≤ 0.
- The depth of the stack is limited by the value of the parameter "max".

(3) EQUIVALENCES

legality(PUSH(T, x)) =

Condition	Value
length(T) = max	%full%
length(T) < max	%legal%

PUSH(T▲, x) = T.PUSH(*, x)

legality(POP(T)) =

Condition	Value
T = _	%empty%
T ≠ _	%legal%

POP(T▲) = T1 where T1: <stack>; x: <int> (T = T1.PUSH(*, x))

legality(TOP(T)) =

Condition	Value
T = _	%empty%
T ≠ _	%legal%

Comments:

- The function *length* is built-in. It returns the number of events in the trace being its argument.
- The argument being defined is tagged with "▲"; hence the notation "PUSH(T▲, x) = Q" means: "If we extend the canonical trace T by the event PUSH(*, x), then Q will be the resulting new canonical trace".
- There is no extension function specified for TOP, since the invocation of this access-program does not change a state of the object at all; Arg#1 of TOP has a single descriptor V (cf. the Syntax Section).

(4) RETURN VALUES

TOP(T)▲ = x where T1: <stack>; x: <int> (T = T1.PUSH(*, x))

Comments:

The value returned by the invocation of TOP ("TOP(T)▲"), x, is equal to the second argument of the last PUSH in the canonical trace T.

Appendix B

Integer Stack Project Guide

IMPLEMENTATION LANGUAGE

Pascal

BUILT-IN TYPES

<int> $\stackrel{df}{=}$ integer.

Comments:

The defined correspondence between the built-in abstract type, <int>, and the concrete type, integer, of the chosen programming language, Pascal, implies that the corresponding abstraction function is known in all internal design documents of the project and can be used implicitly for type conversion.

Integer Stack Internal Design Document

Informal Introduction

This is an internal design document of the module *Integer Stack* specified in Appendix A.

(0) CHARACTERISTICS

- abstract type: <stack>
- features: decentralized, parameterized
- parameters: max: integer

Comments:

In a 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 data structure type is exported from the module.

(1) DATA STRUCTURE

DECLARATIONS

```
EXPORTED type stack = record
                        t: integer;
                        els: array[1..max] of integer
                      end;
```

INITIAL VALUES

init_cds: stack → boolean

init_cds(s) $\stackrel{df}{=}$ s.t = 0

CONSTRAINTS

wfds: stack → boolean

wfds(s) $\stackrel{df}{=}$ $0 \le s.t \le max$

Comments:

- The identifier "s" is a mathematical variable (an argument of a predicate) and not a programming variable.
- The predicate *init_cds* defines initial values of the data structure.
- The predicate *wfds* ("well formed data structure") constrains the set of states of the internal data structure. It constitutes the representation invariant. To verify that the internal design is consistent, it has to be proven that this invariant holds.

(2) ABSTRACTION FUNCTION

af: stack → <stack>

af(s) $\stackrel{df}{=}$ $[PUSH(*, s.els[i])]_{i=1}^{s.t}$

Comments:

The predicate *wfds* need not be stated explicitly in the descriptions of the abstraction function and program functions. We assume that the constraints will hold in all states encountered.

(3) PROGRAM FUNCTIONS

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Value Type
PUSH	<stack>:VO	<int>:V	
POP	<stack>:VO		
TOP	<stack>:V		<int>

ppf_PUSH(s, x) $\stackrel{df}{=}$

	‘s.t < max	‘s.t = max
s’.t =	‘s.t + 1	‘s.t
s’.els |	s’.els[‘s.t + 1] = ‘x ∧ ∀i: integer (1 ≤ i ≤ ‘s.t ⇒ s’.els[i] = ‘s.els[i])	s’.els = ‘s.els

ppf_POP(s) $\stackrel{df}{=}$ s’.els = ‘s.els ∧ (‘s.t = 0 ⇒ s’.t = 0) ∧ (‘s.t > 0 ⇒ s’.t = ‘s.t − 1)

ppf_TOP(s) $\stackrel{df}{=}$ s’ = ‘s ∧ (‘s.t > 0 ⇒ ◣ = ‘s.els[‘s.t])

Comments:

- When " \| " is used in the definition of the "value after", the entries in that row must be boolean expressions; the value of the variable must satisfy the predicate described in the relevant column.
- The value returned by the invocation of TOP is denoted by ◣.
- The argument s in the definitions of the program functions stands for a couple (‘s, s’) where ‘s and s’ denote, respectively, the states of the stack's data structure before and after a program's invocation.

Appendix C

Consistency with the trace specification

It should be proven that our internal design of the stack module is consistent with the interface specification. In Section 4.1 we mentioned conditions that must be satisfied to assure this consistency. In this paragraph we show these conditions as logical formulae. They constitute proof obligations for the internal design.

We start with the program PUSH. First condition addresses the fact that this program preserves the data structure constraint. For each value of the argument and for each state of the data structure that satisfy the constraint (*wfds*) and the legality condition of invocations (*legality* function is not equal to %fatal%) the next state satisfies the constraint (*wfds*). Since the legality function operates on traces, we have to use the abstraction function (af) to map concrete states of the data structure to the abstract states (canonical traces).

$$\forall x\text{: integer; `s, s'}\text{: stack } (\mathit{wfds}(\text{`s}) \wedge \mathit{legality}(\text{PUSH(af(`s), x)}) \neq \%\text{fatal}\% \wedge \text{ppf\_PUSH(s, x)} \Rightarrow \mathit{wfds}(\text{s'}))$$

The second condition is about the existence of the next state. For each value of the argument and for each state of the data structure that satisfy the constraint (*wfds*) and the legality condition of invocations there has to be at least one final state that satisfies the transition relation (ppf_PUSH).

$$\forall x\text{: integer; `s: stack } (\mathit{wfds}(\text{`s}) \wedge \mathit{legality}(\text{PUSH(af(`s), x)}) \neq \%\text{fatal}\% \Rightarrow \exists s'\text{: stack (ppf\_PUSH(s, x))})$$

The third condition expresses the fact that the diagram from Section 4.1 commutes for the program PUSH. For each value of the argument and for each pair of states (`s, s') of the data structure that satisfy the constraint (*wfds*), the legality condition of invocations and the transition relation (ppf_PUSH), transition `s $\rightarrow$ s' must be consistent with the extension function from the interface specification.

$$\forall x\text{: integer; `s, s'}\text{: stack } (\mathit{wfds}(\text{`s}) \wedge \mathit{wfds}(\text{s'}) \wedge \mathit{legality}(\text{PUSH(af(`s), x)}) \neq \%\text{fatal}\% \wedge \text{ppf\_PUSH(s, x)}$$

$$\Rightarrow \text{af(s')} = \text{PUSH(af(`s)}\blacktriangle\text{, x))}$$

Analogous formulae can be constructed for remaining programs (POP, TOP).

The abstraction function should map initial values of the data structure to the empty trace:

$$\forall s\text{: stack } (\mathit{init\_cds}(\text{s}) \Rightarrow \text{af(s)} = \_)$$

This condition constitutes another proof obligation.

Appendix D

Let us consider a problem of inverting a list of $n \geq 1$ integers, a_1, …, a_n, i.e., we want a program such that for every $1 \leq i \leq n$ we have a_i' = 'a_{n-i+1}.

(1) A solution to this problem is presented as a Pascal procedure declaration and its invocation. It is the invocation that must satisfy the specification. The procedure declaration is preceded by definitions and declarations of needed constants, types and variables, to set up the data structure whose values form the state space. In particular, the following assumptions are made about the correspondence between the description of the problem and Pascal programming language entities:

- integer numbers are represented by values of the standard Pascal type integer,
- the length of the list is represented by the constant max ≥ 1,
- the list itself is represented by the value of the variable A of the type

  vector = array[1..max] of integer,

- the phrase Integer Stack(stack, max) introduces an instance of the parameterized specification "Integer Stack"; the name of the type defined by this instance is stack, the value of the parameter (capacity of the stack) is equal to the value of constant max. Thus programs PUSH, POP and TOP are known and their specification is not necessary on displays (cf. Section 5.1),

(2) The following observations and conventions are related to the data state:

- initially, the data state is determined by the values of variables n and A; in the specifications of subprograms we have new variables (actual parameters): size and V,
- each program function (expressed as a relation R_i) specifies acceptable changes of these values (however constants, by definition, do not change and their values need not be mentioned). If the names of the formal arguments of R_i are clear from the context, they may be omitted,
- each competence set (expressed as a set C_i) identifies those values of the data state for which the termination of the program is guaranteed,
- the stack s in the procedures PushDown and PopUp, used to reverse the vector V, is assumed to be empty at the beginning and is required to be empty at the end.

LEXICON

```
const
  max = 1000;
type
  vector = array[1..max] of integer;
  Integer Stack(stack, max);
```

DISPLAY 1

Specification

Reverse(A, n)	
$\mathbf{R}_1(,) = 1 \le n \le max \Rightarrow (\forall i$: integer $(1 \le i \le n \Rightarrow A'[i] = `A[n-i+1]) \wedge NC(n))$	

Program

Procedure declaration:

```
procedure Reverse(var V: vector; size: integer);
var s: stack;
begin
  if size <= max then begin
    PushDown(V, s, size);
    PopUp(V, s, size)
  end
end {Reverse}
```

Specifications of Subprograms

PushDown(V, s, size)	
$\mathbf{C}_2() = (`s = \_ \wedge 1 \le `size \le max)$	
$\mathbf{R}_2(,) = (`s = \_ \wedge 1 \le `size \le max) \Rightarrow (s' = [PUSH(*, `V[i])]_{i=1}^{`size} \wedge NC(V, size))$	

(on Display 2)

PopUp(V,s,size)	
$\mathbf{C}_3() = 1 \le `size \le max \wedge \exists a_1,\ldots,a_{`size}$: integer $(`s = [PUSH(*, a_i)]_{i=1}^{`size})$	
$\mathbf{R}_3(,) = (1 \le `size \le max) \Rightarrow$ $\forall a_1,\ldots,a_{`size}$: integer $(`s = [PUSH(*, a_i)]_{i=1}^{`size} \Rightarrow$ $(s' = \_ \wedge \forall i$: integer $(1 \le i \le `size \Rightarrow V'[i] = a_{`size-i+1}) \wedge NC(size)))$	

(on Display 3)

END OF DISPLAY 1

DISPLAY 2

Specification

PushDown(V, s, size)	
$\mathbf{C}_2()$ = (‘s = _ ∧ 1 ≤ ‘size ≤ max)	
$\mathbf{R}_2(,)$ = (‘s = _ ∧ 1 ≤ ‘size ≤ max) ⇒ (s’ = $[\text{PUSH}(*, \text{‘V}[i])]_{i=1}^{\text{‘size}}$ ∧ NC(V, size))	

Program

Procedure declaration:

```
procedure PushDown(var V: vector; var s: stack; size: integer);
var i: integer;
begin
  for i := 1 to size do
    PUSH(s, V[i])
end {PushDown}
```

Specifications of Subprograms

Empty

END OF DISPLAY 2

Comments:

If it is clear from the context that the programming variables are a, b, c, . …, then one may write “**R**(,)” instead of “R((‘a, ‘b, ‘c, …),(a’, b’, c’, …))”.

DISPLAY 3

Specification

PopUp(V, s, size)	

$\mathbf{C}_3()$ = 1 ≤ ‘size ≤ max ∧ $\exists a_1,\ldots,a_{\text{‘size}}$: integer (‘s = [PUSH(*, a_i)]$_{i=1}^{\text{‘size}}$)

$\mathbf{R}_3(,)$ = (1 ≤ ‘size ≤ max) ⇒
$\forall a_1,\ldots,a_{\text{‘size}}$: integer (‘s = [PUSH(*, a_i)]$_{i=1}^{\text{‘size}}$ ⇒
⇒ (s’ = _ ∧ $\forall i$: integer (1 ≤ i ≤ ‘size ⇒ V’[i] = $a_{\text{‘size}-i+1}$) ∧ NC(size)))

Program

Procedure declaration:

```
procedure PopUp(var V: vector; var s: stack; size: integer);
var i: integer;
begin
  for i := 1 to size do begin
    V[i] := TOP(s); POP(s)
  end
end {PopUp}
```

Specifications of Subprograms

Empty

END OF DISPLAY 3

Appendix E

DI_FID_ADI Module

Informal Introduction

The Attitude Director Indicator is a display in the aircraft cockpit. An ADI device displays an elevation and an azimuth displacement from a fixed reference point.

S_AZIMUTH_INDICATOR(z) causes the indication of azimuth angle z on the ADI.

S_ELEV_IN_VIEW(b) called with b = true causes the ADI elevation indicator to be displayed at the location set by the last call to S_ELEV_INDICATOR. Called with b = false removes the indicator from view. The indicator will remain out of view until the next call to this program with b = true.

S_ELEV_INDICATOR(e) raises %ADI elev not available% if ADI elevation is not available, and sets the ADI elevation indicator to e otherwise.

Note: The prefix S_ (SET) is used for programs that affect the future operation of the module, and the prefix G_ (GET) is used for programs with output parameters.

(0) CHARACTERISTICS

• type specified: <attitude>

• foreign types: <angle>

• features: single-object

• parameters: azimuth_min, azimuth_max, elevation_min, elevation_max: <angle>

Comments:

The DI_FID_ADI module implements only one object of type <attitude>. Since identity of this object is fixed, it need not be passed as argument of access-programs. However, in order to maintain a uniform notation for single-object and multiple-object modules, we write this object explicitly as the zeroth argument.

(1) SYNTAX

INPUT VARIABLES

Variable name	Type	Condition of interest	Event
ELEV_AVAIL	<bool>	true	EVENT_ELEV_AVAIL

Comments:

- An input variable event can be either unconditional or conditional. An unconditional input variable event is used to inform the module about any change of value of this variable. A conditional input variable event is used to inform the module about a change of value of this variable such that the new value satisfies a given condition. A condition of interest for an unconditional input variable event has the form of logical constant true.
- An event name is used in traces to identify an input variable event.

OUTPUT VARIABLES

Variable name:	Type:
ELEV	<angle>

ACCESS-PROGRAMS

Program Name	Arg#0	Arg#1
G_AZIMUTH_INDICATOR	<attitude>:V	<angle>:O
S_AZIMUTH_INDICATOR	<attitude>:VO	<angle>:V
G_ELEV_AVAIL	<attitude>:V	<bool>:O
G_ELEV_INDICATOR	<attitude>:V	<angle>:O
S_ELEV_INDICATOR	<attitude>:VO	<angle>:V
G_ELEV_IN_VIEW	<attitude>:V	<bool>:O
S_ELEV_IN_VIEW	<attitude>:VO	<bool>:V

(2) CANONICAL TRACES

$$\begin{aligned}\text{canonical}(T) \Leftrightarrow \exists e, z{:}\langle\text{angle}\rangle \exists a, d, m, n, v{:}\langle\text{int}\rangle (T = &[\text{S\_AZIMUTH\_INDICATOR}(*, z)]_{i=1}^{a} \cdot \\ &[\text{EVENT\_ELEV\_AVAIL}(*)]_{i=1}^{m} \cdot [\text{S\_ELEV\_INDICATOR}(*, e)]_{i=1}^{d} \cdot \\ &[\text{S\_ELEV\_IN\_VIEW}(*, \text{true})]_{i=1}^{v} \cdot [\text{EVENT\_ELEV\_AVAIL}(*)]_{i=1}^{n} \wedge \\ &0 \le a \le 1 \wedge 0 \le v \le d \le m \le 1 \wedge 0 \le n \le d)\end{aligned}$$

AUXILIARY FUNCTIONS

eavail: <attitude> → <bool>

eavail(T) $\stackrel{df}{=}$ *count*(T,"EVENT_ELEV_AVAIL") = 1

InBoundsAz: <angle> → <bool>

InBoundsAz(x) $\stackrel{df}{=}$ azimuth_min ≤ x ≤ azimuth_max

InBoundsEl: <angle> → <bool>

InBoundsEl(x) $\stackrel{df}{=}$ elevation_min ≤ x ≤ elevation_max

ElevInView: <attitude> → <bool>

ElevInView(T) $\stackrel{df}{=}$ *count*(T, "S_ELEV_IN_VIEW") = 1

(3) EQUIVALENCES

legality(G_AZIMUTH_INDICATOR(T, n)) =

Condition	Value
count(T,"S_AZIMUTH_INDICATOR") = 0	%FID not set%
count(T,"S_AZIMUTH_INDICATOR") = 1	%legal%

legality(S_AZIMUTH_INDICATOR(T, z)) =

Condition	Value
¬*InBoundsAz*(z)	%FID display bounds%
InBoundsAz(z)	%legal%

S_AZIMUTH_INDICATOR(T↘, z) =

Condition	Value
count(T,"S_AZIMUTH_INDICATOR") = 0	S_AZIMUTH_INDICATOR(*,z).T
count(T,"S_AZIMUTH_INDICATOR") = 1	S_AZIMUTH_INDICATOR(*,z).T1 where T1: <attitude>; z1: <angle> (T = S_AZIMUTH_INDICATOR(*,z1). T1)

legality(G_ELEV_AVAIL(T, n)) = %legal%

legality(G_ELEV_INDICATOR(T, n)) =

Condition	Value
¬*eavail*(T)	%ADI elev not available%
eavail(T) ∧ *count*(T,"S_ELEV_INDICATOR") = 0	%ADI elev not set%
eavail(T) ∧ *count*(T,"S_ELEV_INDICATOR") = 1	%legal%

legality(S_ELEV_INDICATOR(T, e)) =

Condition	Value
¬*InBoundsEl*(e)	%FID display bounds%
InBoundsEl(e) ∧ ¬*eavail*(T)	%ADI elev not available%
InBoundsEl(e) ∧ *eavail*(T)	%legal%

S_ELEV_INDICATOR(T↘, e) =

Condition	Value
count(T,"S_ELEV_INDICATOR") = 0	T.S_ELEV_INDICATOR(*,e)
count(T,"S_ELEV_INDICATOR") = 1	T1.S_ELEV_INDICATOR(*,e).T2 where T1,T2: ≪attitude≫; e2: <angle> (T = T1.S_ELEV_INDICATOR(*,e2).T2)

legality(G_ELEV_IN_VIEW(T, n)) =

Condition	Value
¬*eavail*(T)	%ADI elev not available%
eavail(T)	%legal%

legality(S_ELEV_IN_VIEW(T, d)) =

Condition	Value
¬*eavail*(T)	%ADI elev not available%
eavail(T) ∧ *count*(T,"S_ELEV_INDICATOR") = 0	%ADI elev not set%
eavail(T) ∧ *count*(T,"S_ELEV_INDICATOR") = 1	%legal%

S_ELEV_IN_VIEW(T↘, d) =

Condition	Value
ElevInView(T) ⇔ d	T
¬*ElevInView*(T) ∧ d	T.S_ELEV_IN_VIEW(*,true)
ElevInView(T) ∧ ¬d	T1 where T1: <attitude> (T = T1.S_ELEV_IN_VIEW(*, true))

EVENT_ELEV_AVAIL(T↘) =

Condition	Value
∃!T1: <attitude> (T = T1.EVENT_ELEV_AVAIL(*))	T1
¬∃T1: <attitude> (T = T1.EVENT_ELEV_AVAIL(*))	T.EVENT_ELEV_AVAIL(*)

(4) RETURN VALUES

G_AZIMUTH_INDICATOR(T, n↘) = z where z: <angle>; T1:<attitude>
(T = S_AZIMUTH_INDICATOR(*,z).T1)

G_ELEV_AVAIL(T, n↘) = *eavail*(T)

G_ELEV_INDICATOR(T, n↘) = e where e: <angle>; T1,T2: ≪attitude≫
(T = T1.S_ELEV_INDICATOR(*,e).T2)

G_ELEV_IN_VIEW(T, n↘) = *ElevInView*(T)

OUTPUT VARIABLES

ELEV(T) =

Condition	Value
¬*eavail*(T) ∨ ¬*ElevInView*(T) ∨ *count*(T,"S_ELEV_INDICATOR") = 0	
eavail(T) ∧ *ElevInView*(T) ∧ *count*(T,"S_ELEV_INDICATOR") = 1	e where e: <angle>; T1,T2: ≪attitude≫ (T = T1.S_ELEV_INDICATOR(*, e).T2)

Comments:
≪attitude≫ denotes the set of all traces of type attitude, while <attitude> consists only of canonical traces.

Appendix F

DI_FID_ADI Project Guide

IMPLEMENTATION LANGUAGE

Pascal

BUILT-IN TYPES

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

<int> $\stackrel{df}{=}$ integer

DI_FID_ADI Internal Design Document

(0) CHARACTERISTICS

- abstract type: <attitude>
- foreign types: <angle>
- features: single-object, centralized
- parameters: azimuth_min, azimuth_max, elevation_min, elevation_max: <angle>

Comments:

In the centralized internal design of the single-object module there is only one object data structure and it belongs to the module. The sole object is created by the module before any invocations of its access programs.

(1) DATA STRUCTURE

DECLARATIONS

az, e: <angle>;

ev_nb, az_nb, ei_nb, eiv_nb: integer;

INITIAL VALUES

init_mds: <angle> × <angle> × integer × integer × integer × integer → boolean

init_mds(az, e, ev_nb, az_nb, ei_nb, eiv_nb) $\stackrel{df}{=}$ ev_nb = 0 ∧ az_nb = 0 ∧ ei_nb = 0 ∧ eiv_nb = 0

CONSTRAINTS

wfds: <angle> × <angle> × integer × integer × integer × integer → boolean

wfds(az, e, ev_nb, az_nb, ei_nb, eiv_nb) $\stackrel{df}{=}$ $0 \le az\_nb \le 1 \wedge 0 \le eiv\_nb \le ei\_nb \le 1 \wedge 0 \le ev\_nb \le 1 \wedge$

$(az\_nb = 1 \Rightarrow InBoundsAz(az)) \wedge (ei\_nb = 1 \Rightarrow InBoundsEl(e))$

OUTPUT VARIABLES

ELEV: <angle>;

(2) ABSTRACTION FUNCTION

af: <angle> × <angle> × integer × integer × integer × integer → <attitude>

af(az, e, ev_nb, az_nb, ei_nb, eiv_nb) $\stackrel{df}{=}$ [S_AZIMUTH_INDICATOR(*, az)]$^{az\_nb}$.

.[EVENT_ELEV_AVAIL(*)]$^{ev\_first}$. [S_ELEV_INDICATOR(*, e)]$^{ei\_nb}$.

.[S_ELEV_IN_VIEW(*, true)]$^{eiv\_nb}$. [EVENT_ELEV_AVAIL(*)]$^{ev\_last}$

where ev_first, ev_last: integer ((ev_nb = 1 ⇒ (ev_first = 1 ∧ ev_last = 0)) ∧

((ev_nb = 0 ∧ ei_nb = 1) ⇒ ev_first = ev_last = 1) ∧

((ev_nb = 0 ∧ ei_nb = 0) ⇒ ev_first = ev_last = 0))

(3) PROGRAM FUNCTIONS

INPUT VARIABLES

Variable name	Type	Condition of interest	Event
ELEV_AVAIL	<bool>	true	EVENT_ELEV_AVAIL

ppf_EVENT_ELEV_AVAIL $\stackrel{df}{=}$ NC(az, e, az_nb, ei_nb, eiv_nb) ∧

	‘ev_nb = 0	‘ev_nb = 1
ev_nb’ =	1	0

ACCESS-PROGRAMS

Program Name	Arg#0	Arg#1
G_AZIMUTH_INDICATOR	<attitude>:V	<angle>:O
S_AZIMUTH_INDICATOR	<attitude>:VO	<angle>:V
G_ELEV_AVAIL	<attitude>:V	<bool>:O
G_ELEV_INDICATOR	<attitude>:V	<angle>:O
S_ELEV_INDICATOR	<attitude>:VO	<angle>:V
G_ELEV_IN_VIEW	<attitude>:V	<bool>:O
S_ELEV_IN_VIEW	<attitude>:VO	<bool>:V

ppf_G_AZIMUTH_INDICATOR(azim) $\stackrel{df}{=}$ NC(az, e, ev_nb, az_nb, ei_nb, eiv_nb) ∧

	‘az_nb = 1	‘az_nb = 0
azim’ =	‘az	

ppf_S_AZIMUTH_INDICATOR(azim) $\stackrel{df}{=}$ NC(e, ev_nb, ei_nb, eiv_nb) $\wedge$

	InBoundsAz(‘azim)	¬*InBoundsAz*(‘azim)
az_nb’ =	1	‘az_nb
az’ =	‘azim	‘az

ppf_G_ELEV_AVAIL(b) $\stackrel{df}{=}$ NC(az, e, ev_nb, az_nb, ei_nb, eiv_nb) $\wedge$ b’ = (‘ev_nb = 1)

ppf_G_ELEV_INDICATOR(elev) $\stackrel{df}{=}$ NC(az, e, ev_nb, az_nb, ei_nb, eiv_nb) $\wedge$

	‘ei_nb = ‘ev_nb = 1	‘ei_nb = 0 $\vee$ ‘ev_nb = 0
elev’ =	‘e	

ppf_S_ELEV_INDICATOR(elev) $\stackrel{df}{=}$ NC(az, ev_nb, az_nb, eiv_nb) $\wedge$

	InBoundsEl(‘elev) $\wedge$		¬*InBoundsEl*(‘elev)
	(‘ev_nb = 1)	(‘ev_nb = 0)	
ei_nb’ =	1	‘ei_nb	‘ei_nb
e’ =	‘elev	‘e	‘e

ppf_G_ELEV_IN_VIEW(b) $\stackrel{df}{=}$ NC(az, e, ev_nb, az_nb, ei_nb, eiv_nb) $\wedge$

	‘ev_nb = 1	‘ev_nb = 0
b’ =	‘eiv_nb = 1	

ppf_S_ELEV_IN_VIEW(d) $\stackrel{df}{=}$ NC(az, e, ev_nb, az_nb, ei_nb) $\wedge$

	‘ev_nb = 1 $\wedge$			‘ev_nb = 0
	(‘ei_nb = 1) $\wedge$		(‘ei_nb = 0)	
	d	¬d		
eiv_nb’ =	1	0	‘eiv_nb	‘eiv_nb

Comments:

In a single-object module the zeroth argument of an access-program constitutes the module’s data structure and thus can be omitted from the argument list of the corresponding program function.

OUTPUT VARIABLES

ELEV =

ev_nb = 1 $\wedge$ eiv_nb = 1 $\wedge$ ei_nb = 1	ev_nb = 0 $\vee$ eiv_nb = 0 $\vee$ ei_nb = 0
e	