EDITOR FOR THE TRACE ASSERTION METHOD

Michal Iglewski[a], Marcin Kubica[b], Jan Madey[b]

[a] Département d'informatique, Université du Québec à Hull, Hull, Québec, J8X 3X7, Canada

[b] Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland

ABSTRACT

In this paper we describe a project aiming in development of software tools supporting creation of documentation needed in the process of designing, implementation and verification of computer systems.

1. Functional approach to computer systems documentation

Documentation for computer systems should be analogous to the precise and detailed documentation produced in other areas of engineering. Essential properties of computer systems and their components can be described by a set of mathematical relations. By providing these relations, computer system designers can document their designs systematically, and use that documentation to conduct thorough reviews. This approach can be applied to all phases of computer system design [6, 7]. We call it "functional" since the basic mathematical concepts - functions and relations, play a key role.

The complete documentation of a software system must include a system requirements document and a system design document, which together determine a software requirements document. We should also have a software module guide, which describes the structure of the software system. For each module identified in the module guide, there should be a module interface specification. For every implementation of a module interface specification (there may be several), we should document the module internal design; that document describes the internal data structures and the effect of the module's access-programs on the state of that structure. The contents of these documents are defined in [6, 7], which contain a more general discussion of the role and structure of functional documentation in software engineering.

2. FUN-TOOLS project

The above described form of computer systems documentation has been used in a number of practical situations, cf [1, 5]. The experience has confirmed the well-known fact that support tools are definitely needed before we can expect an extensive use of the functional approach. There are several reasons of that:

- In theory, the documentation can also be used for mathematical verification of the design. In practice, the documentation was used for a systematic manual verification only; tools would assist both applications.
- The relations used in functional documentation are often conveniently represented using a tabular form of expressions, in which the table entries are themselves expressions; such notation is quite readable but difficult to produce manually.
- Creation of some documents is a hard task when using LaTeX or some desktop publishing software, like

FrameMaker (many special symbols, nested tabular forms, etc.); more sophisticated support is needed.

- Mechanical validation of consistency between some documents and within them is needed; proper tools are indispensable for that.

FUN-TOOLS is a joint project between Université du Québec à Hull, Canada and Warsaw University, Poland. The project started at the beginning of 1993. The principal objectives are:

- Integration of different documents in the functional model of documentation of computer systems.
- Application of the functional approach to practical examples.
- Development of tools supporting the production of different documents and verification of the design process.

The basic tool components of FUN-TOOLS project are:

(a) FUN-SPEC (an editor for module interface specifications).

(b) FUN-INTER (an editor for internal design documents, containing a language independent kernel and a family of language dependent editors).

(c) FUN-REQ (an editor for requirements specifications).

(d) FUN-DISP (a system supporting the Display Method of program design and presentation).

In this paper we will not discuss the latter three subprojects but concentrate on FUN-SPEC. Its purpose is to help in the development of module interface specifications expressed in terms of the Trace Assertion Method [4, 2].

3. The Trace Assertion Method

The basic principles of this method (called in short TAM) are as follows. In general case, by a *module* we understand a group of visible *access-programs* using a hidden data structure. A module can be viewed as implementing one or more finite state machines called *objects*. All communication between the outside world and the objects implemented by a given module is achieved via:

- a vector of external state variables that the object "observes" (the *input variables*),
- a vector of variables that are controlled by the object and can be observed externally (the *output variables*), and
- the module's *access-programs* that can be used by other modules to provide information to the object, and/or receive information from it.

The values of the input variables are considered to be observable at any time. To observe them the module may use hardware devices or externally supplied programs. Actual observations are made at points in time determined by the environment.

A state change in the object may be caused only by an external *event*, i.e. an invocation of an access-program, or a change in the values of the input variables.

A *trace* is a sequence of events affecting the object. If an event is an access-program invocation, the trace includes also those output values (returned by this access-program) which influence the module's future behavior. The set of traces is partitioned into a finite number of equivalence classes according to the equivalence relation mentioned below. Each equivalence class is represented by one of its elements called

the *canonical trace.* Intuitively, canonical traces correspond to states of the module. The set of canonical traces is characterized by the predicate, called *canonical*.

A complete "black-box" description of a module in the trace assertion method is described by:

- functions, that map from single event extensions of canonical traces to the equivalent canonical traces; these functions - called *extension functions* - define the *equivalence relation* on traces,
- a relation, that maps each canonical trace to a vector of sets of values of output variables,
- relations, that associate each single (access-program) event extension with a set of returned values for each output argument.

A module interface specification will contain the following sections:

(a) Characteristics Section.

(b) Syntax Section.

(c) Canonical Trace Section.

(d) Equivalence Section.

(e) Return Value Section.

The Characteristics Section states whether the module specification is parameterized or not, single-object or not, deterministic or not. This section also lists the type defined by the module, foreign types used by the module, possible specification parameters, etc.

The Syntax Section contains an access-program table. The access-program table contains one row for each access-program, one column for each argument, and an additional column for the value returned by the program. Each entry specifies the type of the corresponding argument and characterizes its role.

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, possible additional notation for abstract values, and the lexicon of other notational abbreviations.

The Equivalence Section describes the extension functions for access-programs invocations. These functions defines the equivalence relation on traces.

The Return Value Section defines the values returned by access-programs.

4. The Synthesizer Generator

One of the basic assumptions of our project is that the set of tools has to be well integrated. A tool for specification of module interfaces will interfere with tools for system requirements specification and module internal design. In particular, a module internal design can be viewed as an implementation of a module interface specification. It means, that the module internal design has a common part with the module interface specification. A generation of that part should be supported by the developed tool. We have decided to use the Synthesizer Generator (previously called the Cornell Program Synthesizer) [8] as a development platform for the FUN-TOOLS project, because it provides all necessary mechanisms and it significantly reduces the time needed to develop an editor. There were also some other factors which determined our choice of this tool, in particular: a friendly user interface, a support for structural and textual editing, and possibilities of expressing and checking semantical and logical properties.

The Synthesizer Generator (in short: SG) is a tool for implementing language-based editors. The edited text is internally represented as a derivation tree of the language's abstract syntax. This tree can be also decorated with a number of attributes. With the help of these attributes, the contextual and semantical conditions can be expressed quite easily. If some of these conditions are violated, attributes can be also used to display errors. Every time the text is changed, only a subtree containing that text is updated. An incremental analysis is performed throughout the tree to update all attributes depending on the modified text. The way in which the derivation tree (edited text) and calculated attributes are displayed on the screen, is defined by the editor's designer. One can define several ways of displaying the edited text. They can be used, for instance, to list all contextual errors found in the text or to generate a LaTeX document.

Another very useful feature of SG are transformations. A transformation is composed of two parts: a pattern and a replacement value. A transformation is enabled if its pattern matches the value of the structural selection. The replacement value is a function of the selected term. The replacement is done on the user's request. Transformations can be used for simple actions (e.g. to insert or replace a symbol) or to invoke a more time-consuming computation.

The editor designer has to prepare:

- a context-free grammar representing language's abstract syntax,
- language's concrete input syntax,
- rules describing the way(s) in which the abstract syntax is displayed on the screen,
- equations defining the values of the attributes,
- patterns and replacement values of transformations.

The Synthesizer Generator creates a "ready to run" editor according to these rules. No other information is needed to generate an editor. In case of very simple editors, attributes and transformations can be omitted.

5. The FUN-SPEC editor

The editor implements the version of TAM described in [2]. The development of the editor required a precise codification of the method's syntax and elements of its semantics. It has revealed a number of syntactical and semantical ambiguities in the existing TAM reports [4, 2]. Also, with the help of the editor we discovered a number of mistakes in examples, which have been read by many people. A revised version of TAM based on our work with FUN-SPEC is described in [3].

One of the design principles was to minimize the amount of information needed to be given by a user of the editor. While the document is being prepared, its latter sections can be partially generated based on the previous ones. The module interface specification is naturally split into two parts, the first one containing syntax declarations, and the second one in which the semantical properties are described. Therefore the form of the second part depends on the contents of the first part. The skeleton of the second part can be generated automatically, based on the first part. Already this single feature of the editor significantly reduces the amount of work needed to fill in the Equivalence Section and the Return Value Section.

The editor helps also during the development phase of the both parts. One should notice that certain modifications in the first part can imply that some elements of the second part can totally change their meaning or loose it temporarily (e.g. as a result of renaming module's access-programs or modifying their arguments' declarations). We have defined some strategies for dealing with such situations.

The incremental character of the editor and the redundancy in specifications create a big challenge. A modification of a feature in one place should be consistent with other features, and possibly, imply changes in these places. A lot of contextual and semantical conditions (e.g. type correctness) can be checked on-line. The problem has been solved by the usage of nonterminal attributes [8] and by choice, in such situations, of a responsible part of the specification. Errors can be reported in the appropriate places in the specification, as soon as an inconsistency appears.

Transformations can be used to select a particular form of a trace, logical or set expression, to apply an operator (e.g. to replace $\neg\forall$... by $\exists$...) or a logical law, or to invoke a more time-consuming operation, like validation of some logical conditions. For example, in tabular expressions (which are widely used in the functional documentation), several correctness conditions have to be satisfied. In the example expression:

Condition		Equivalence
$(T = \_) \vee (U = \_)$		%no_init%
$(T \neq \_) \wedge (U \neq \_) \wedge ((nbElems(T) + nbElems(U))$	> xmax)	%out_of_space%
	$\leq$ xmax)	... expr ...

all conditions have to be mutually exclusive and the disjunction of them have to cover the whole domain. Transformations can be helpful in several ways. In particular, they can be used to split and join the rows of the table without changing its correctness. For example, a transformation can generate a new row in the table, with the condition equal to the negation of the condition in the row above. Another transformation can generate two subrows, for two results of the inequality. Combining such transformations we can both reduce the time needed to fill in the conditions and keep the correctness conditions of the table valid. As a side effect, generated conditions are sometimes slightly redundant.

We can also provide several “logic-law” transformations. By applying them one can evaluate conditions to the form, for which the correctness of the table is obvious. This kind of transformations is usually simple to implement. For example, to implement one of the de Morgan’ laws, we need to provide the rule saying that the formula: $(\alpha \wedge \beta)$ can be converted into $\neg(\neg\alpha \vee \neg\beta)$. Such a transformation can involve some more complex computations, like the transformation of the formula to the clause form.

The editor can display the specification in several views. Every view is a collection of prettyprinting rules. The basic one is used to edit the specification text in a roughly formatted form. Some other views are used when an object is saved to a file in text format; the ASCII view is used as a primary storage format, and the LaTeX view generates the text in the form of a LaTeX document. Some additional functions are provided to view or print the document. Such an interface can be build also for different desktop publishing software (e.g for FrameMaker), but each interface would require a separate view. There is also a view containing only all contextual errors found in the specification. All one has to do to localize a specified error in the text is to put the cursor on the error message. The cursor will move to the corresponding locations in all other views.

6. Future plans

Our short-term objectives include: an introduction of input variable events, creation of a library of specifications, design and implementation of a set of prototype tools intended to support the production and use of tabular documentation.

Our long-term objectives include: a better integration of different documents in the functional model, reuse of existing specifications (locating, comparing and selecting, specialization, integration, introduction of object-oriented paradigms), tools for "black-box" testing, incorporating a theorem prover.

7. Conclusions

The editor under development within the FUN-SPEC subproject is a part of tools supporting the functional approach to the documentation of the module interface specification. Some of its features, like incremental edition, checking semantic correctness, friendly user interface give us a powerful tool which allows a user to concentrate on essential issues instead of technical details. The editor has been used in classrooms for a few months. It helped to find errors in specifications existing for a long time and studied by many readers.

Acknowledgements

The authors greatly appreciate the contributions of Bernard Desruisseaux and Patrick Roy in the work on the editor. Special thanks are due also to Janina Mincer-Daszkiewicz and Krzysztof Stencel for discussions on this project and comments on previous versions of this article.

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), and by DEC's European External Research Programme (EERP).

References

1. Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E., "Software Requirements for the A-7E Aircraft", *NRL Memorandum Report 3876*, United States Naval Research Lab., Washington D.C., November 1978, 523 pp.
2. Iglewski, M., Madey, J., Parnas, D.L., Kelly P. C., "Documentation Paradigms", *CRL Report 270*, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada; July 1993, 45 pp.
3. Iglewski, M., Kubica, M, Madey, J., Stencel, K., "Towards a Formal Definition of the Trace Assertion Method", in preparation.
4. Parnas, D.L., Wang, Y., "The Trace Assertion Method of Module Interface Specification", *Technical Report 89-261*, Queen's University, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, October 1989, 39 pp.
5. 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.
6. Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering. (Version 2)", *CRL Report 237*, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada; September 1991, 14 pp.
7. 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.
8. *The Synthesizer Generator Reference Manual, Rel. 4.1*, GrammaTech, Inc., One Hopkins Place Ithaca, NY.