An Easily Extensible Toolset for Tabular Mathematical Expressions

David Lorge Parnas[1] and Dennis K. Peters[1,2]

[1] Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. John's, Newfoundland
Canada A1B 3X5
dpeters@engr.mun.ca
[2] Department of Computing and Software
Faculty of Engineering
McMaster University
Hamilton Ontario
Canada L8S 4K1

Abstract. We describe the design of an open ended set of tools for manipulating multi-dimensional tabular expressions. The heart of the toolset is a set of modules that makes it possible to add new tools to the toolset without having detailed knowledge of the other tools. This set of modules, the Tool Integration Framework, supports new and existing tools by providing abstract communication interfaces. The framework that we describe has proven to be a practical approach to building an extensible set of tools.

1 Introduction

The McMaster University Software Engineering Research Group (SERG) studies documentation methods for computer systems engineering that use mathematical functions and relations [18] to describe program behaviour. The mathematical expressions that describe the behaviour of real systems are often so complex that they are difficult to write and use. When expressions are written in a tabular form, they are much more easily formulated and interpreted [16].

While the value of tabular notation has often been demonstrated [4, 5, 3, 6, 11], we believe that well designed tools can reduce both the effort needed to write tabular expressions and the number of errors in the documentation. To demonstrate this, our research group is developing a suite of tools, collectively known as the Table Tool System (TTS), for creating, editing, printing, analysing and interpreting tabular documentation. This paper presents an overview of the design of the TTS. It is intended both to draw attention to the TTS and to provide an example of a system in which modularisation, abstraction, and other related design principles are applied consistently.

New components of the TTS are usually produced as Masters Theses by students who come, learn about software engineering, and then leave. It is normally extremely difficult to get such independently written components to work together in a useful way. The Tool Integration Framework that is described in this paper has changed that. Components can be developed independently, then easily integrated into our system by people who do not know the details of either the old or the new components.

1.1 Background

Much traditional engineering documentation is mathematically based and precise, rich in information content and consistently interpreted from user to user. Parnas and Madey [18] have shown how the essential properties of computer systems can be described by using mathematical relations. These relations can be characterised by first-order predicate logic (e.g., [17]]. By providing these relations, computer systems designers can document their designs systematically, and use that documentation to conduct thorough reviews and analysis.

The expressions that characterise the relations that result from applying functional documentation techniques to real programs are usually complex because they must distinguish many cases. When represented in their customary form (i.e. as a one dimensional expression) they would be too difficult to read to be practical. In [16], Parnas defined a new notation, called *tabular expressions*, that grew from earlier work at the Naval Research Laboratory in Washington, DC [4]. Tabular expressions in this new form have the same theoretical expressive power as conventional notation but, by organising the expression as an array of much simpler expressions, they are much easier for human readers to interpret. The reader is referred to [16, 7, 6] for descriptions of table types and interpretations.

The importance of readability in engineering documentation is clear when the role of documentation in the design process is considered. Documentation should capture the measurable objectives of the design effort. The actual results of the design effort can be compared with the objectives expressed in the documentation at several points during the design process. For example, documentation will be used as the basis for design reviews [13] and as the basis for testing procedures.

The examples in Fig. 1 show three different formats for an expression representing the function $f(x, y)$. The benefits of representing expressions in tabular form are shown even more clearly by longer, more realistic, expressions such as those in [1, 23].

Without support tools, a great deal of time is spent performing tasks that could be automated. Mathematical expressions can be checked mechanically and used for automated verification of the design specified by the documents. For example, manually comparing two tables to see if they represent the same function is a very important task that can be very time consuming and tedious; small errors can be difficult to detect. We need tools to help automate those jobs that can be automated, so that our time and energy can be devoted to the more interesting tasks of system design.

$$f(x,y) = \begin{cases} 0 & \text{if } x \geq 0 \wedge y = 10 \\ x & \text{if } x < 0 \wedge y = 10 \\ y^2 & \text{if } x \geq 0 \wedge y > 10 \\ -y^2 & \text{if } x \geq 0 \wedge y < 10 \\ x+y & \text{if } x < 0 \wedge y > 10 \\ x-y & \text{if } x < 0 \wedge y < 10 \end{cases}$$

(a) the function $f(x,y)$

$$\forall x, \forall y, \begin{pmatrix} ((x \geq 0 \wedge y = 10) \rightarrow f(x,y) = 0) \wedge ((x < 0 \wedge y = 10) \rightarrow f(x,y) = x) \wedge \\ ((x \geq 0 \wedge y > 10) \rightarrow f(x,y) = y^2) \wedge ((x \geq 0 \wedge y < 10) \rightarrow f(x,y) = -y^2) \wedge \\ ((x < 0 \wedge y > 10) \rightarrow f(x,y) = x+y) \wedge ((x < 0 \wedge y < 10) \rightarrow f(x,y) = x-y) \end{pmatrix}$$

(b) $f(x,y)$ described using classical predicate logic

$f(x,y) =$

	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	x	$x+y$	$x-y$

(c) $f(x,y)$ described using tabular notation

Fig. 1. Example Representations of the Function $f(x,y)$

1.2 Goals

Because the syntax of tabular expressions is not compatible with existing document production tools such as word processors, symbolic mathematics processors, spreadsheets, etc., the goal of the TTS project is to develop an integrated, extensible system of tools—that is, a set of tools that work together; to facilitate the use of tabular expressions, e.g., in computer systems documentation.[1] Many complex tasks can be accomplished relatively easily by combining tools from a small set of carefully designed components. TTS capabilities include such things as entering and modifying tabular expressions, long term storage of expressions on disk, formatting of expressions for output, and transforming tables to other forms.

Our understanding of how such systems should be implemented is growing. The development of TTS, and its need for extensibility, reflects this process. Each tool is first developed in a basic form and refined as experience with its use teaches us how to better implement it. Ideas for completely new components will certainly arise in the future. It is important to have a stable interface standard that defines how these tools should interact with one another so that tools developed at different times by different designers can operate with each other. It is important to establish these standards as early in the project as practical to minimise the amount of rework.

[1] "Working together" means, for example, that an input tool can pass a tabular expression to an output tool for display or to an evaluation tool to calculate a value.

Each new member of our group typically either develops some new component to extend the functionality of the TTS, or tries to apply the TTS to solve a new problem. To accomplish this, it is essential that the TTS be easy to extend, even if the extension had not been foreseen, and that the TTS components can be easily combined in new ways. Since people join the group at different times, work at different rates, have different backgrounds and usually leave as soon as they finish their thesis, we cannot expect them to function as a cohesive development team. Our tool integration framework seeks to overcome this by allowing developers to work independently while taking full advantage of the work done by others.

1.3 General Design Principles

The Table Tool System is designed for future growth by encapsulating design decisions regarding interaction between TTS components in a common set of modules known as the Tool Integration Framework.

The modules of the TTS all have a “hidden” implementation [14]. By using the principle of information hiding, users of a module need not know anything about the internal data structures and other implementation details of that module. In other words, any module can be treated as a black box. The interface to a module consists a set of *access programs*. TTS developers can only access modules by calling those programs. The use of any TTS module actually consists of writing a list of access program calls, for example, to create, manipulate and destroy objects of a type implemented by the module. Modules may support any number of objects, so in the currently popular object-oriented terminology, each implements an *object class* where the access programs correspond to *methods*. Although inheritance can save effort and time during the initial coding, we have seen it used in ways that make subsequent maintenance more difficult and have avoided it. We did not find it necessary to use an O-O language or terminology although we applied many of the good design principles that are implicit in some O-O approaches.

Since we are planning for a large family of TTS tools, we have not followed the classical “top down” approach to system design. Instead, we have chosen to specify and construct some basic “building-blocks” to be used in the whole tool family. These building-blocks have been selected using the principle that E. W. Dijkstra has called “separation of concerns”.

1.4 Documentation

Since our research focus is documentation for computer systems development, it makes sense to use this project as a proving ground for our methods. We have found that a combination of both formal and informal documents is required. The complete documentation of the TTS is given in [24].[2] The informal system

[2] The printed form of this document is not kept up to date. For internal use an up-to-date electronic version is maintained.

overview and module guide sections, from which this paper is primarily drawn, serve to introduce new group members to the structure and capabilities of the TTS. In addition, for each module, we produce two different styles of interface description: an informal module interface guide, and a formal module interface specification. The informal document serves as a quick guide to the module so that developers can gain an intuitive understanding of its capabilities. The formal specification is used as a reference document to get specific information about the module behaviour.

For the kernel modules, which are critical to the system, formal internal design documents have been produced as well and these have been used for structured review of the code. Unfortunately, since the TTS was in its early stages of development when the kernel was built, we could not use it to help with that documentation process, so we learned first hand how tedious it can be to use mathematical methods without supporting tools.

2 TTS Module Structure

This section describes the modular structure of the TTS by describing the secret (hidden information) of each module in the system. Figure 2 shows a design schematic for the TTS: nested boxes represent the sub-module relationship, and the vertical arrangement roughly corresponds to the uses hierarchy.[12, 15] At the first level of decomposition, the TTS is divided into three modules: kernel, infrastructure and applications, which are described in more detail below. The intended relationship between these modules is that programs in the kernel module use only other kernel programs, while programs in the infrastructure module use other infrastructure programs or kernel programs, but not applications, and applications may use any program. Thus, these modules can be viewed as tiers, each building on the ones below. These tiers do not, however, necessarily correspond to levels in the uses hierarchy since, for example, each module may contain programs that use no others, and hence are at the lowest level of the hierarchy. The tiers structure gives a useful overview of the system, but it omits information that is included in the uses relation, which is needed, for example, for maintenance and testing.

Kernel The lowest tier contains essential modules—those that are used by all other TTS modules. It hides the implementation of the abstract data types that represent expressions, their semantics and representation. These data types, known as *TTS objects*, are the only objects that may be passed between tools. Kernel modules do not interpret expressions.

Infrastructure The middle tier contains tools that operate on TTS objects to provide some service to the user (e.g., modification of an expression), and modules that allow these tools to be combined. These are the 'blocks and mortar' from which applications are constructed. The infrastructure provides a useful set of operators for kernel objects but does not share the secret of the kernel.

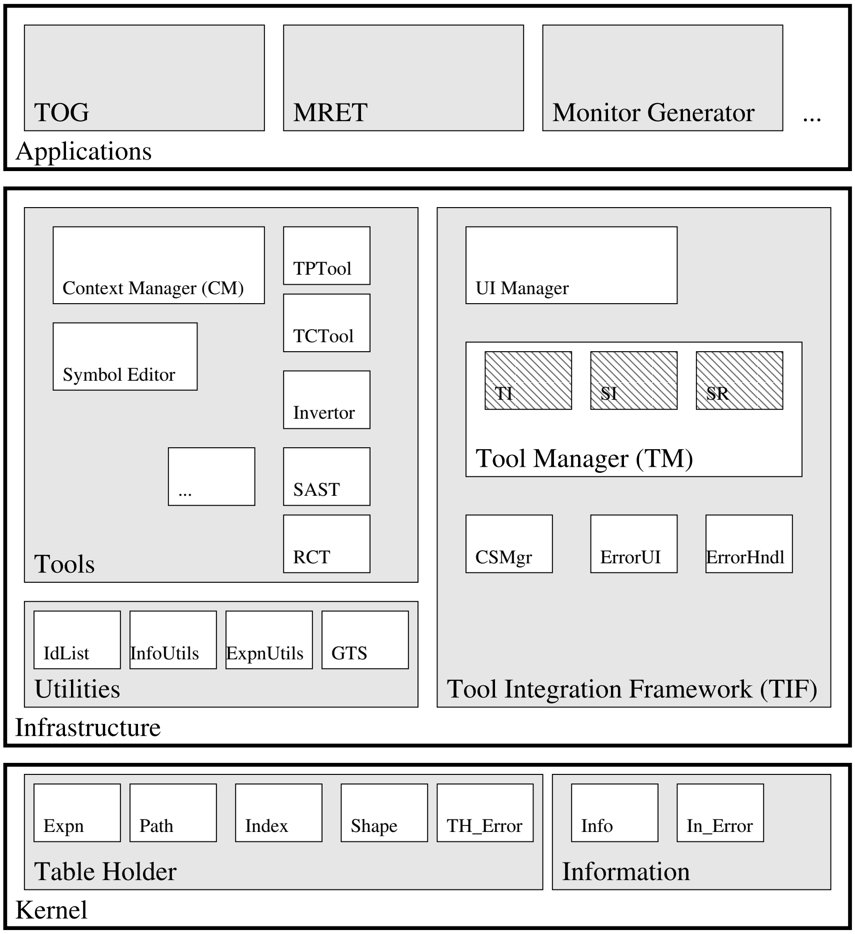

Fig. 2. TTS Design Schematic

Applications The top tier contains programs that combine infrastructure modules to manipulate or interpret groups of tabular expressions as documents, which specify or describe some aspect of a computer system. For example, a set of expressions might be interpreted as a relational specification of the intended behaviour of a program, and an application could be used to check if the observed behaviour of the program is consistent with the specification.

2.1 Kernel

The kernel module hides data structures representing expressions, and algorithms for manipulating them. The kernel is divided into two modules:

table holder hides data structures representing expression structure and identifiers;

information module hides the means of associating presentation and semantics information with particular symbols used in expressions.

Table Holder A mathematical expression, whether tabular or not, can be viewed as a set of components joined by operators, where each component is itself an expression. The tools that interpret these expressions must be able to access any component of the expression. The table holder module provides a mechanism that allows tools to do this without knowing how the information about the expression is represented within the computer.

Each component of an expression has an address, which is the path that one must follow to reach that component from a starting point that is the address of the whole expression. Each path is a sequence of indices; each index identifies a subexpression. Each index in the path brings us closer to the subexpression addressed by the path.

Information stored in the table holder is restricted to the structural information about expressions; this is the information needed by all tools that manipulate expressions. It contains no information about interpretation. For example, a table is stored as an array of grids without any assumptions about which grid will be considered the main grid and which are to be interpreted as headers. A function is stored as a list of arguments and an identifier which will index into an external semantic description. By carefully isolating these elements of the expressions, which are independent of interpretation and are not likely to change, the table holder forms the foundation upon which the table tools system is built. The table holder contains four sub-modules: Expression, which hides the data structure representing actual expression structure, Shape, Index and Path, which each hide data structures representing auxiliary objects that facilitate the expression manipulation activities that must be performed. These are explained below.

Expression The Expression module hides the implementation of the data structure and algorithms for representing and manipulating expression syntactically. Expressions are grouped into three different categories based upon their different structures; atoms (constants or variables), applications (functions or predicates with 1 or more arguments), and tables. The module has access programs for creating, destroying, copying, modifying, reading, comparing, loading and saving expressions.

Index The Index module hides the data structure for representing the position of a table cell within a table. An `Index` consists of a grid number (`int`) and a sequence of n numbers (`int`), where n is the dimensionality of the grid, which indicate a position in the table (e.g. the row number and column number of the cell). The module has access programs for creating, copying and destroying indices, as well as assigning and retrieving grid numbers and dimension values and incrementing or decrementing an index value with respect to a `Shape`.

Path The Path module hides the data structure for representing the position of a sub-expression within an expression. A `Path` is a sequence of objects, each of which is either an `Index` or an `int`. Integers are used to specify a particular argument of an application and `Index` objects identify an element of a table. The length of a `Path` can be increased by inserting an element, or decreased by deleting an element at any position. The module has access programs for creating, copying and destroying paths, as well as inserting, assigning, deleting and retrieving elements at any position of the path and retrieving the length of a path.

Shape The *shape* of a table is the number of grids it contains and the number of dimensions and length in each dimension of each grid. The Shape module hides data structure for representing the shape of a table. It exists so that a user can separate the task of describing the shape of a table from the action of creating the table. A `Shape` object can be retrieved from an existing table when needed. The module has access programs for creating, copying and destroying shapes, as well as assigning and retrieving the number of grids, the dimensionality of each grid and the length of each grid in each dimension.

Information The information module associates each `Id` stored in a table holder expression with information about the symbol that it represents. The secrets of the Information module are the data structures used to associate an `Id` with its data and the algorithms for manipulating these. The symbols are grouped into symbol tables within which each symbol is identified by an `Id`, the value of which is determined by the information module when the symbol is created. The `Id` can be stored as part of an expression in the table holder so that other modules can use it to gain access to the data in the information module. For each symbol table, the information about symbols is organised into named information classes (e.g. name, type, font family). Each symbol may or may not have data for a given information class, so that new classes can be added when needed—for example, to support new tools—without needing to modify existing symbols, expressions or tools. This is key to the extensibility of the TTS.

The module provides access programs for creating and destroying symbol tables and information classes, and for creating symbols and assigning or retrieving data for a particular class `name` and symbol `Id`, or searching to find symbols that are associated with specific data or data patterns.

2.2 Infrastructure

The TTS infrastructure module hides data structures and algorithms that enable users to manipulate TTS objects. This module contains three modules:

Tool Integration Framework TIF modules provide a means of interaction between tools and between users and the TTS.

Tools *Primitive services* are atomic operations on expressions or symbols that the user can invoke from the TTS main menu (e.g., printing, combining or editing expressions). A *tool* is a module that provides one or more primitive services.

Utilities A *utility* is a module that implements an algorithm that may be useful to more than one tool and is independent of both the user interface and the tool integration framework. Utilities cannot be invoked directly by a user (i.e. they do not provide 'primitive services').

Tool Integration Framework (TIF) The TIF module hides the interface between tools so that they need not know what other tools are in use or available in the system. It also provides a user interface for invoking individual tools and passing objects between them in a consistent manner. It is made up of five modules, as described below.

Tool Manager (TM) The secrets of the tool manager are the data structures that represent those characteristics of tools that are relevant to the TIF, and the information needed about the run-time instances of tools. The TM also hides the algorithms for invoking the tools. The tool manager defines a language that can be used to define applications by describing combinations of tools. Available components can be added or deleted at run-time. The TM defines a standardised interface, to which all of the tool modules it manages must conform. It is composed of three sub-modules; the Service Registry, the Instance Registry, and Tool Interface modules.

A *service* is a procedure that the user invokes via a single command. It is provided by executing a sequence of one or more operations (*primitive services*) provided by the TTS tools. The Services Registry contains the following information about each service, so that it can be invoked when needed:

- the type of user interface required (i.e., graphical, none),
- what name and menu item is used to refer to the service,
- what type(s) of objects a service operates on,
- how the service is provided (i.e., what access programs, of what tools, in what order).

The Instance Registry module tracks service use, updating instances of primitive services executing at a particular time. The Tool Interface module maintains information about the invocation history of the tools so that it can determine what needs to be done before it can handle requests for service (e.g. does a tool need to be initialised first etc.).

User Interface Manager (UIMgr) The UIMgr provides a UI 'framework' in which tools can interact with users without their designers needing to invest effort developing UI functionality that is not specific to their tool—it hides the characteristics of Motif that would otherwise need to be known by the developer of each tool that has a graphical user interface (e.g., it encapsulates the Motif event loop). For tools that have a graphical UI, the UIMgr makes available a parent window handle, so that the tools can create their own windows as children of the parent window. It also provides a uniform means for tools to inform other tools of significant events (e.g., an expression has been modified) without the sender or receiver needing any knowledge of the other tool(s). UIMgr does not hide the choice of Motif as a UI platform, however, since to develop a sufficiently general purpose abstract interface for graphical user interfaces would involve duplication of a significant portion of systems such as Motif.

The secret of the UIMgr is the implementation of the TTS 'top level' User Interface, which enables users to invoke services to operate on TTS objects and to switch between concurrently executing services. It also hides the calls to access programs that initialise the TIF and kernel modules and the order in which they are called.

User Error Interface The secret of the Error User Interface module is the method of reporting error and status messages to users. It provides access programs for tools to report errors to users. The user can customise the UI so that errors of severity level below a threshold will not be reported (although not all error messages can be disabled) and can have error messages logged to a file.

Error Handling The Error Handling module hides the data structure for representing the status of invocations of tools and TIF module access programs. It also hides an algorithm for translating a status token into a textual description. The module has access programs for setting the status as well as for retrieving the status token or the textual description of a status token.

Clipboard/Selection Manager (CSMgr) The Clipboard/Selection Manager module hides data structures and algorithms for passing TTS objects between tools. It defines a standard interface through which TTS objects can be exchanged between tools.

Tools Module The Tools module hides algorithms and data structures for tools that provide primitive services enabling users to directly manipulate TTS objects. Some example tools are described below. Tools and applications are normally composed of several sub-modules but, since these are not the focus of this paper the detailed structure is not described here.

Context Manager (CM) A *context* is an ordered set of of named expressions together with a symbol table containing information about the symbols in the expressions. Since the expressions use the same symbol table, Ids will have consistent interpretation throughout the context, which simplifies manipulation and interpretation of expressions.

The CM hides the implementation of a user interface and file format for working with contexts. Several contexts may be in use at the same time within one application. The user interface allows the user to

– load a context from disk,
– save a context to disk,
– create new (empty) expressions,
– delete expressions, and
– select expressions for use by other tools.

Table Printing Tool (tptool) The Table Printing Tool module hides the implementation of a tool for viewing and changing the physical appearance of an expression without changing its meaning. The tool allows the user to adjust such things as the print size of parts of expressions, and the width and height of table rows or columns, but does not allow any modifications to the contents of the expression, or changes to the symbol information that may change their interpretation (e.g. font-face).

The tool produces a Postscript representation of the expression, suitable for printing. It uses the TH and info modules, respectively, to retrieve the expression and symbol information.[26]

Table Construction Tool (TCT) The TCT hides the implementation of a user interface for entering and editing the contents of an expression, while ensuring that it is syntactically correct. It allows the user to construct an expression by building it up from smaller expressions. It uses the information module to retrieve symbols and the TH to retrieve the expression. Several instantiations of the TCT may be in use at the same time within one application.[10]

Specialisation and Simplification Tool (SAST) The specialisation and simplification tool provides algorithms for simplifying tabular expressions by taking into account user-supplied constraints on the variables that appear in the expression.

Symbol editor The Symbol Editor allows a user to modify the set of symbols available for use in expressions. It consists of two sub-modules; the Symbol Editor UI and the Symbol Utilities modules.

The Symbol Editor UI module hides the implementation of a user interface for loading, viewing, selecting, editing and saving the information about symbols.

The secrets of the Symbol Utilities module are the files and information classes used to represent common (default) symbols and symbol property inheritance in the Information module. It has access programs that mirror some of the access programs of the Info module but take default symbols and inheritance into account.

Table Inverter The table inverter module hides algorithms for 'inverting' and 'normalising' tabular expressions. In some cases, tabular expressions can be easier to understand, or made more compact if displayed in a different form. Table transformations and tools for performing these functions are described in [22].

Table Composition Tool The table composition tool hides algorithms to calculate the relational composition of two tabular expressions.[25]

Table Checking Tool The table checking tool hides algorithms to check, using an automated proof system, that tabular expressions satisfy two conditions: disjointness and completeness, which are usually requirements for correct specifications.[8].

Utilities The Utilities module hides algorithms that may be useful to more than one tool but do not implement primitive services and hence cannot be invoked directly by a user. This module is independent of the user interface system (i.e. Motif) and the TIF module interface.

Kernel Utilities Kernel Utilities are utilities that make use of the TTS kernel.

The expression utilities module hides algorithms for traversing the sub-expressions of an expression and for manipulating the set of `Id`s used in an expression. It has access programs for calling a caller-supplied program for each sub-expression of an expression, for finding all `Id`s used in an expression and for substituting all occurrences in an expression of an `Id` from one list with the corresponding `Id` in another list.

The info utilities module hides algorithms for performing common operations on symbol tables that have been created by the info module. It has access programs for merging tables, removing lists of `Id`s from tables and finding the intersection or union of the set of classes in two tables.

The generalised table semantics module hides algorithms and data structures that represent the semantics of a table as part of an expression. This semantic information can be accessed by other TTS table evaluation tools and applications like the test oracle generator. (See 2.3)

General Utilities General utilities are those utilities that do not make use of the TTS kernel.

The secret of the id list module is the data structure for representing sequences (lists) of `Id`s. It also hides algorithms for searching and manipulating these lists. Note that since this module does not manipulate `Id`s in any way, it is independent of the TTS kernel.

2.3 Applications

The applications module hides the implementation of applications, which treat TTS objects as components of relational documentation and allow the user to edit, analyse or interpret that documentation.

Test Oracle Generator (TOG) The test oracle generator interprets a set of expressions as a program specification and uses it to generate a ‘test oracle’, which can be used to verify the actual behaviour of a program against the specification.[19, 21]

Module Reliability Evaluation Tool (MRET) The module reliability estimation tool interprets a set of expressions as a module interface specification and, using an operational profile and a module under test, estimates the reliability of the module.[9]

Monitor Generator The monitor generator interprets a set of expressions as a system requirements specification for a real-time system and uses it to generate a 'monitor', which reports if the system behaviour is consistent with the requirements.[20]

3 Experience

In order to test the viability of the TIF and to validate our design, several tools that were developed prior to its conception have been integrated into the TTS. The integration process in some cases has been complicated by inconsistencies between the various developers in the assumptions implicit in their designs. Despite this, we have been successful in integrating these tools, and more tools are being added as resources permit.

Group members who have started to develop new tools since the TIF has been added have found that they have a significant advantage over their predecessors. They do not have to spend time developing support software in order to demonstrate their results.

Although the TIF is a relatively new addition to the TTS, our experience so far has convinced us of its value and of the suitability of the 'framework' architecture to development environments such as ours. We have found that the architecture encourages new group members to develop tools that integrate smoothly with the TTS and, at the same time, allows them to concentrate on the research problem at hand by removing the need for them to create support software in order to demonstrate their results. Although it is difficult to be certain with such a small sample size, it appears that since the introduction of the TIF to the TTS the pace of tool development and integration has increased significantly.

4 Further Information

Space limitations prevent us from providing complete descriptions of the interfaces to these modules. Further information can be found in [24] or by visiting the TTS web page at `http://www.crl.mcmaster.ca/SERG/TTS/ttsHome.html`.

Acknowledgements

The TTS project has had a long life and has benefited from the contributions of many people, some of whom are named in the references. In particular we acknowledge the contributions of Ramesh Bharadwaj, an early architect, and project engineers Kalpana Bharadwaj, Carl Krasnor, Jim Hare and Ruth Abraham.

The comments and suggestions from the anonymous referees have helped us to improve this paper.

Funding for this work is provided by the Natural Sciences and Engineering Research Council (NSERC), Bell Canada, and Communications and Informations Technology Ontario (CITO).

References

[1] Brian J. Bauer. Documenting complicated programs. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, December 1995. Also printed as CRL Report # 316, Telecommunications Institute of Ontario.

[2] *Proc. Conf. Computer Assurance (COMPASS)*, Gaithersburg, MD, June 1995. National Institute of Standards and Technology.

[3] Constance L. Heitmeyer, A. Bull, C. Gasarch, and Bruce G. Labaw. SCR*: A toolset for specifying and analyzing requirements. In COMPASS '95 [2], pages 109–122.

[4] Katherine Heninger, David Lorge Parnas, John E. Shore, and J. Kallander. Software requirements for the A-7E aircraft. Technical Report MR 3876, Naval Research Laboratory, 1978.

[5] S. D. Hester, D. L. Parans, and D. F. Utter. Using documentation as a software design medium. *Bell System Technical Journal*, 60(8):1941–1977, October 1981.

[6] Douglas N. Hoover and Z. Chen. Tablewise, a decision table tool. In COMPASS '95 [2], pages 97–108.

[7] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science—Advances in Computing Science*, pages 184–196. Springer Wien, New York, 1997.

[8] Min Jing. Checking table tool. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, to appear 1998.

[9] ChunMing Li. Software reliability estimation tool. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, December 1996. Also printed as CRL Report # 337, Telecommunications Institute of Ontario.

[10] Weimin Li. Table construction tool. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, July 1996. Also printed as CRL Report # 330, Telecommunications Institute of Ontario.

[11] D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April-June 1991.

[12] David L. Parnas. On a 'buzzword': Hierarchical structure. In *Proc. IFIP Congress*, pages 336–339. North Holland, 1974.

[13] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. In *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 132–136, August 1985.

[14] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, pages 1053–1058, December 1972.

[15] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979.

[16] David Lorge Parnas. Tabular representation of relations. CRL Report 260, Communications Research Laboratory, November 1992.

[17] David Lorge Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[18] David Lorge Parnas and Jan Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[19] Dennis K. Peters. Generating a test oracle from program documentation. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, April 1995.

[20] Dennis K. Peters. *Deriving Real-Time Monitors from System Requirements Documentation.* PhD thesis, McMaster University, Hamilton ON, to appear 1999.

[21] Dennis K. Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.

[22] H. Shen, J. I. Zucker, and D. L. Parnas. Table transformation tools: Why and how. In *Proc. Conf. Computer Assurance (COMPASS)*, pages 3–11, Gaithersburg, MD, June 1996. National Institute of Standards and Technology.

[23] Hong Shen. Implementation of table inversion algorithms. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, December 1995. Also printed as CRL Report # 315, Telecommunications Institute of Ontario.

[24] Software Engineering Research Group. Table tool system developer's guide. CRL Report 339, Communications Research Laboratory, January 1997.

[25] Albert H. Tyson. Function composition tool. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, August 1998. Also printed as CRL Report # 364, Telecommunications Institute of Ontario.

[26] Li Zhang. A template/overlay approach to displaying and printing tables. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, June 1994. Also printed as CRL Report # 289, Telecommunications Institute of Ontario.

10.1007/b107031130024