Design and Documentation of the Kernel of a Set of Tools for
Working With Tabular Mathematical Expressions

Darren John Bane
Software Quality Research Laboratory
University of Limerick

19 July 2008
ABSTRACT

Given a new mathematical model for tabular expressions, a reusable kernel that mechanises this model is designed and documented. This kernel design supports upper-level table tools by providing facilities to define table types, store tabular expressions and evaluate tabular expressions. Disciplined software development methods are applied to the design of the kernel. For example the principle of information-hiding is applied and module interfaces are formally specified using the Trace Function Method.
ACKNOWLEDGEMENTS

I would like to thank Prof. David L. Parnas for his guidance and help throughout my Masters degree. Without his guidance and comments much of the work would not be in its current form. In addition, I wish to thank Dr. Gerard O’Regan, Dr. Ying Jin, Dr. Shimata Tshinanga and Dr. Tom Arbuckle for the constant review of my work and insightful comments and aid.

I am grateful for the privilege of working on interesting new research, contributing to the new mathematical model of tabular expressions and being an early adopter of the Trace Function Method.

I wish to acknowledge the financial support of the Science Foundation Ireland under SFI grant 01/P1.2/C009 for the duration of this research.

Finally, I would like to thank my friends and family for their support and patience, especially my wife Emma.
# Table of Contents

The Table Tool System ................................................. 1  
What are mathematically rigorous documents? ..................... 1  
  What are conventional expressions? .......................... 2  
  What are tabular expressions? ................................ 2  
  What are the contents of mathematically rigorous documents? .. 3  
What is difficult about writing, checking, printing and evaluating them? .. 4  
How can the TTS help? .............................................. 4  

The Functions of the kernel ........................................ 6  
The role of the kernel .............................................. 6  
  Defining table types ...................................... 6  
  Storage of expressions .................................. 7  
  Evaluating expressions .................................. 7  
  Checking the restriction ................................. 8  
  Storing reusable definitions ......................... 8  

Previous work .................................................... 9  
What tabular expression tools were previously written? .......... 9  
What did the earlier tools do and how do they differ from ours? .. 9  
  Briefly, how was the McMaster TTS structured? ............ 10  
  How were table types defined in other tools? .............. 10  
  How does the McMaster TTS define table types? .......... 11  
  How does HTables define table types? .................. 12  
  How were expressions stored in other tools? .............. 13  
  How were functions defined for reuse in other tools? ...... 14  
  How was evaluation performed by other tools? ............ 14  
What else did the previous tools do? .......................... 16  
  What else did the McMaster tools do? .................. 16  
  What else did the SCR* tools do? ........................ 16  
  What did the OPGI tools do? ............................. 17  
What tabular expressions does the kernel specification support? .. 18  
Why do we need a semantics for tabular expressions? .......... 18
What decisions are we not hiding? 35
What design decisions did we make? 36
How are cross-cutting concerns, such as storage, implemented in a module hierarchy? 36
How does this structure facilitate the use of previously-existing software to provide some of the services in Section 2? 38
What is the detailed internal structure of our module interface documents? 39
What sections are in all interface specifications? 39
  System generation parameters 39
  Local set definitions 40
  Output variables 40
  Input variables 40
  Access programs 40
  Trace functions dictionary 41
  Output variable functions 41
How was our error reporting policy specified? 42
How do we describe a design to make resource management more deterministic? 42
  reserve 43
cancelRes 43
Example 43
How might the TTS kernel be implemented? 45
Which software tools (compilers, libraries, etc.) were considered and which are recommended? 46
Development environment 46
What language constructs are suitable for implementing the concept of modules as defined in Parnas (1972) and how do they compare to each other? 46
What rules were used to choose names in the documentation? 47
How could error handling and memory management be implemented? 48
  Error handling 48
  Logging 49
  Memory management 49
  Concurrency 50
How can the services offered by the TTS kernel (as described in Section 2) be used
by client software? .......................... 51
How can errors be handled? .......... 51
How do we begin a transaction? .......... 52
How do we end a transaction? .......... 52
How do we define symbols in the library? .......... 53
How is a conventional expression stored? .......... 54
How are expressions evaluated? .......... 55
How can the tasks of storage and evaluation be performed for tabular expres-
sions? ............................................. 56
Conclusion ........................................ 59
What was learned? .................. 59
  What was learned about specifying systems? .......... 59
  What was learned about specifying systems using the TFM? .......... 60
  What structural conventions for software documentation were discov-
ered? ............................................. 60
  What was learned about applying the SCR techniques to the domain of
symbolic processing rather than the more usual real-time/embedded control? .......... 61
  What improvements to the method and extensions to the software are worthy
of further investigation? .......... 61
  How can documentation be improved? .......... 61
  Briefly, are there any complementary approaches to improving soft-
ware quality? ........................................ 62
  How would we change the design of the kernel? .......... 63
  How would we change how the kernel was designed? .......... 63
References ........................................ 65
1. The Table Tool System

This thesis documents the kernel of a set of tools for documenting software, and outlines the steps taken and conventions adopted to design it.

This section introduces the Table Tool System (TTS), of which the kernel is a part. Section 2 defines the scope of the kernel in the TTS. Section 3 discusses similar previous work. Section 4 introduces a new mathematical model of tabular expressions. Section 5 discusses some of the issues that were encountered. Section 6 outlines the documentation techniques that were found appropriate to design the kernel. Section 7 describes how the kernel was decomposed into modules. Section 8 describes the conventions used when documenting the module interfaces. Section 9 shows one way that our documentation can be translated into an implementation. Section 10 discusses the kernel features from the point of view of tool writers using the proposed kernel implementation. Finally, section 11 gives the conclusions and suggests some directions for future work.

Rigorous design through documentation is considered desirable for many problem domains. It is believed that this thesis demonstrates that it is also feasible.

1.1 What are mathematically rigorous documents?

Current industrial practice is that if software is documented at all it is usually documented in natural language. It has long been recognised that natural language is often ambiguous, incomplete, poorly organised and inconsistent.

“... I found the inadequacy of language to be an obstacle; no matter how unwieldy the expressions I was ready to accept, I was less and less able, as the relations became more and more complex, to attain precision ...”

Frege (1879).

Program code allows much less nondeterminism than is in typical requirements specifications so many decisions are deferred until implementation time. The problem then is that the implementer may not be in communication with the customer, and so what is produced may not be what the customer requires. We want to achieve a lack of ambiguity in the initial documents and a way to check them for completeness with mathematical rigour.

“A proof or demonstration is said to be rigorous if the validity of each step and the connections between the steps is explicitly made clear in such a way that the
result follows with certainty ...

Weisstein (2000).

Many attempts, e.g. (Diller 1994) and (George 2002), have been made over the years to put software documentation on a sound mathematical footing, but none of these have seen wide adoption. This thesis builds on and supports one of these methods, specifying software using relations (Parnas and Madey 1995), in an effort to widen that method’s applicability.

It is important to note that rigour is not the goal here. Efficient communication is the goal and rigour is a means to that end.

1.1.1 What are conventional expressions?

*Conventional expressions* is the term we use for ordinary mathematical expressions, as can be seen in any mathematics, engineering or science textbook. Some examples of such expressions are “x + 2”, “P ∧ Q” and “∀(x, x ∈ N ⇒ ∃(y, y ∈ N ⇒ y = x + 1))”.

We define predicate expressions as the subset of both conventional and tabular expressions that have a boolean value. We interpret predicate expressions according to Parnas (1993).

1.1.2 What are tabular expressions?

Informally, tabular expressions are a generalisation of conventional mathematical expressions. They are a multi-dimensional representation, whereas conventional expressions are all one-dimensional strings of symbols (though the layout is sometimes two-dimensional).

Table 1 shows an example of a tabular expression.

<table>
<thead>
<tr>
<th></th>
<th>x &lt; 0</th>
<th>x = 0</th>
<th>x &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y &lt; 0</td>
<td>x^2 - y^2</td>
<td>x^2 + y^2</td>
<td>x^2 - y^2</td>
</tr>
<tr>
<td>y = 0</td>
<td>x + y</td>
<td>x^2 - y^2</td>
<td>x + y</td>
</tr>
<tr>
<td>y &gt; 0</td>
<td>x^2 + y^2</td>
<td>x + y</td>
<td>x^2 + y^2</td>
</tr>
</tbody>
</table>
In the table type taxonomy of Abraham (1997) and Balaban et al. (2007) Table 1 is classified as a normal table. Table 1 represents the following conventional expression:

\[
\begin{align*}
f(x, y) &= \begin{cases} 
  x^2 - y^2 & \text{if } (((y < 0) \land (x < 0)) \lor ((y < 0) \land (x > 0)) \lor ((x = 0) \land (y = 0))) \\
  x + y & \text{if } (((y = 0) \land (x < 0)) \lor ((y = 0) \land (x > 0)) \lor ((y > 0) \land (x = 0))) \\
  x^2 + y^2 & \text{if } (((y < 0) \land (x = 0)) \lor ((x < 0) \land (y > 0)) \lor ((x > 0) \land (y > 0)))
\end{cases}
\end{align*}
\]

Like conventional expressions, tabular expressions can be evaluated to yield a value. Evaluating a normal table yields the value of the cell with indices of the positions of whatever predicates in the top and left grids that are true. Specifically, the top grid determines the column number and the left grid determines the row number of the expression that will be evaluated to obtain the final value. For example, if we evaluate the above tabular expression at \(x = 1\) and \(y = 0\), this selects the cell at coordinates (3, 2) in the main grid. The expression in this cell, and hence the value of the function that the tabular expression defines, is \(x + y\), i.e. 1.

The normal table type, like other table types, is defined according to Balaban et al. (2007) by two conventional expressions, an evaluation term and a restriction term. Many other table types may be defined. One of the innovations of this proposed Table Tool System (TTS) is that it supports the addition of new table types.

Tabular expressions are formally defined later in this thesis, in Section 4, but for now an informal definition is: a tabular expression is an indexed set of grids, each grid is an indexed set of cells, and each cell contains an expression.

Note: the cells of tabular expressions can contain either tabular or conventional expressions. The expressions in the inner most cells will always be conventional, i.e. not sets of grids.

Tabular expressions are often a more readable way to present conventional expressions. Any tabular expression can be converted to an equivalent conventional expression. Some of the work that a reader performs to parse a long mathematical expression is already done by dividing the expression up into grids and cells.

### 1.1.3 What are the contents of mathematically rigorous documents?

Tabular expressions have been found to be useful and practical in documenting and analysing many aspects of computer systems (Baber et al. 2005; Heninger et al. 1978; Quinn et al. 2006), and that is the intended use of the proposed tools. A software system can be
defined using a set of relations (Parnas and Madey 1995) where output (or controlled) variables are defined as functions or relations on the domain of input (or monitored) variables. These relations, often described in tabular form, are the contents of our mathematically rigorous documents.

1.2 What is difficult about writing, checking, printing and evaluating them?

Word and text processing packages have been able to lay out tables of text for quite some time. However, although these packages can input and lay out tabular expressions, they know nothing about what they mean so even the most trivial of errors can only be found by human review.

Keying in tabular expressions is time-consuming. Word and text processors only help with basic input and formatting, and even then this requires significant manual interaction to get right. Other features, such as spelling checkers, make keying in mathematics even more difficult. Word and text processors can print tables well but it requires tedious work by a human to get the desired results.

Without tool support, evaluation requires a human to manually convert the tabular expressions into conventional form. Many tools exist to evaluate conventional expressions, e.g. Computer Algebra Systems (CAS) such as Maple and Mathematica.

1.3 How can the TTS help?

The proposed TTS is a set of tools to manipulate tabular expressions. Initially included will be an input tool, output tool and kernel but we have tried to design the system for ease of extension and contraction (Parnas 1979) so it should be easy to add more tools later. The input tool will help with the input of tabular expressions. The output tool will lay them out in a format suitable for printing on paper. The kernel is a library that all tools will use to perform common tasks such as storage (in both volatile and persistent memory) and evaluation. In the TTS, the kernel provides facilities for storing, retrieving, evaluating and checking tabular expressions. There are also facilities for defining new types of tables.

Dedicated support for input, evaluation and printing will save the manual effort now required to accomplish these tasks, making the use of tabular expressions more palatable in industrial practice. Tabular expressions in turn make the use of mathematics in documentation more palatable.
Unlike documents written in natural language, limited machine-checking of mathematical documents is feasible. This possibility of machine-checking for syntax and basic semantic errors is a major advantage for mathematically rigorous documents. Detecting these errors early lets human reviewers concentrate on the more serious questions, such as whether the contents of the requirements document match the customer’s requirements. However, such machine-checking requires tools to be written. This is a major motivation for the Table Tool System.
2. The Functions of the kernel

The services that the TTS kernel provides to tools are enumerated in this section†.

2.1 The role of the kernel

There is core functionality common to all tools for processing tabular expressions. The kernel will be a reusable library to perform these tasks.

The kernel provides the following services to calling programs:
1. It stores table type definitions
2. It maintains a library of types, functions and variables that can be used in expressions
3. It stores expressions
4. It checks that a tabular expression meets conditions stated in the definition of the table’s type
5. It evaluates stored expressions, including the storage of assignments of values to variables

Expressions stored in the kernel can be conventional or tabular. All tasks not mentioned above will be performed by tools or other libraries that are not part of the kernel but use it when performing their functions.

The kernel only implements primitive functions, i.e. those that can’t be defined in terms of other kernel functions. Ease of use or convenience is not a primary motivation, since the users of the kernel will be other programs. Where it is useful, a layer of utilities can be implemented above‡ and outside of the kernel and used by many tools.

2.1.1 Defining table types

Before a table of a particular type can be stored (or evaluated) by the kernel, the type must be defined. The definition of new table types is one of the functions that the kernel can be used to perform.

Following the mathematical model (Balaban et al. 2007), we define table types by giving a *restriction term* and an *evaluation term*. Both are expressions that may be either

† This section is largely based on (Parnas 2005).
‡ Higher in the “uses hierarchy” (Parnas 1979)
conventional expressions or tabular expressions of a previously defined table type. The restriction term is a predicate that must always be true for all tables of a particular type (similar to an invariant). The evaluation term defines the meaning of a tabular expression by giving an equivalent expression (for an example, see Section 1.1.2). That expression may be either a conventional expression or an expression of a previously-defined table type.

Table types, the functions with which they are defined and the types of these functions are symbols whose definition is stored in the library (see Section 2.1.5). Note that if the evaluation of library symbols is implemented by some external evaluator, e.g. a CAS, mutually recursive definitions may be unacceptable to that evaluator.

How to define table types is specified as the Sym module in Appendix B.

2.1.2 Storage of expressions

All envisioned tools will need to store tabular and conventional expressions, so this function should be performed by the kernel. The kernel should be the only component in all the tools to perform this function. The storage of both tabular and conventional expressions is supported.

How to store expressions is specified as the Expr and Graph modules in Appendix B.

2.1.3 Evaluating expressions

We define an assignment as a binding of free variables to values. Given such an assignment, we can evaluate an expression, i.e. substitute the values for variable placeholders in the expression and simplify it to a less complex form (Carette 2004). It is an intentional limitation of the TTS kernel that the result of such a simplification be only one object, i.e. partial evaluation is not performed. Because conventional expressions appear in tabular expressions and are a special case of tabular expressions, an evaluator must be able to handle all expressions.

The kernel will provide no built-in functions but instead will provide a mechanism for adding functions. This is to avoid treating functions defined by a user differently from those defined by a tool writer.

The ability to evaluate expressions will be useful to many tools, e.g. a Trace Function Method (TFM) simulator currently under construction at the Software Quality Research Laboratory (SQRL) or a table-checker (Jing 2000). As a result, such functionality should be
provided by the kernel in conjunction with a function library.

How to evaluate expressions is documented as the Eval module in Appendix B.

2.1.4 Checking the restriction

The kernel checks the restriction term of a table for each assignment of values to variables immediately before evaluation. Recall that the restriction term is a predicate that must be true for the table type. Checking the restriction term for the general case of all possible assignments would require theorems to be proven, and is beyond the scope of the kernel. However, if this is done, additional checking by the kernel is not needed and could be bypassed.

2.1.5 Storing reusable definitions

Conventional expressions are composed of function applications (all the standard operators are treated as functions). At any time there are a finite number of such functions and table types available for use by specification writers, although more can always be defined and added. Sets of these available functions and table types are termed libraries in the TTS kernel. Each tabular expression must be associated with exactly one library that determines the semantics of the operators, functions and table types used in the expression. Such libraries are the implementation mechanism by which the kernel can have no built-in functions or operators—there are no special cases.

Enough semantic information must be stored for each operator, function and table type in a library to enable limited checking, evaluation and querying of the list of available constituents for use by a user interface.

Within one library, functions are uniquely identified to the kernel by their name only, i.e. overloading is not supported.

How to store reusable definitions is documented as the Library module in Appendix B.
3. Previous work

3.1 What tabular expression tools were previously written?

The TTS kernel is intended to be the re-usable core code for a set of tools for supporting the preparation and use of tabular expressions. These tools could be used to gather requirements and to document software. This section discusses other projects that performed similar tasks, concentrating on the functions of our separate kernel but also including a broad outline of the systems as a whole.

These tools each focused on a subset of the tasks in a software life-cycle. A brief overview of their capabilities follows, together with an analysis of what can be learned for the TTS kernel.

Details of four toolsets written to help use tables in software development have been published: McMaster University’s Table Tool System (Software Engineering Research Group 1997), HTables (Kahl 2003) from the same institution, the Naval Research Laboratory’s (NRL) tools (Heitmeyer, Archer et al. 2005) and Ontario Power Generation Inc.’s (OPGI) tools (Lawford et al. 2004).

3.2 What did the earlier tools do and how do they differ from ours?

There are significant differences in the capabilities of the tools that these groups delivered in the past, as shown in Table 2.

Table 2. Previous tool capabilities. Key: ✓ provided, ? cannot be determined from published literature, blank not provided.

<table>
<thead>
<tr>
<th>Feature</th>
<th>McMaster TTS</th>
<th>HTables</th>
<th>NRL</th>
<th>OPG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input from specification writer</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Output to paper</td>
<td>✓</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>Checking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Evaluation/simulation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Separate kernel</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
<td>✓</td>
</tr>
</tbody>
</table>

Sections 3.2.2 to 3.2.6 outline how the tools listed in Table 2 perform their functions. If some tools are omitted, this means that I could not find relevant information in the published literature.
3.2.1 Briefly, how was the McMaster TTS structured?

The McMaster TTS is the immediate predecessor of our TTS. It was implemented over a period of many years by the staff and graduate students of the Software Engineering Research Group at McMaster University, Canada (Software Engineering Research Group 1997).

The McMaster table tools were divided into three layers:

1. The Table Holder and Information modules. These manage the data structures that describe a table, both in memory and on disk. This software is used by all tools.

2. The Library layer. This contains functionality which is needed by several (but not all) tools. Example services provided by the library layer are a context manager and a platform abstraction.

3. The tools themselves. These reuse the above application framework. Tools provided included a test oracle generator, reliability estimator, transformation tool and input tool.

The functionality required of the kernel in the current TTS was spread over several places in McMaster’s system as described above.

3.2.2 How were table types defined in other tools?

A tool may have a fixed set of table types that may be supplemented only by expanding the code, or it may contain a definition facility. If the latter, the flexibility is a variable, meaning that there are several levels of support for such definitions, which may make adding new table types much easier or more difficult.

Three possible levels of flexibility in defining table types are:

1. No built-in provision for adding table types. Code must be hand-written to support new table types.

2. A formal model with each table type implemented by separate hand-written code. This differs from the above in that there is a common conceptual model of tables, although the code for each instance of this model is still separate.

3. A formal model and a definition mechanism based on it. Adding new table types using this mechanism must be easier than changing tool source code to do so.
The big disadvantage of (1) or (2) is that the knowledge of table types is built into tools, making these table types “special”, when we believe that they are merely instances of the more general model that have been found useful in the past.

The NRL tools had dedicated editors for each table type, so seem to be at level (1). This was normal practice until Janicki (1997) demonstrated that a more general approach is possible.

**3.2.2.1 How does the McMaster TTS define table types?**

The McMaster TTS supported (2), defining table semantics in terms of the model of Janicki (1997).

In this paper, a header is an indexed set of cells. A cell is treated as a primitive that does not need to be explained. A raw table skeleton is a collection of headers plus a grid indexed “by the headers” according to the paper, which we understand as meaning indexed by the Cartesian product of the index sets of the headers. A Cell Connection Graph (CCG) is a relation between (cells in) the headers and (cells in) the main grid. This is not the relation that the table describes; instead it is a structural artefact of the Janicki model. If we interpret the CCG relation as an directed acyclic graph, each arc is required to either start or end with the main grid. The headers and the grid, together with a CCG, define a medium table skeleton. The domain of the function defined by a tabular expression is defined by a table predicate rule and the range is defined by a table relation rule. A medium table skeleton together with table predicate and relation rules is called a well-done table skeleton. We get a full tabular expression by adding a mapping that assigns a predicate expression to each predicate cell, and a relation expression to each relation cell.

This model may be better illustrated by the example in Table 3 (Janicki 1997).

**Table 3. An example of a raw table skeleton T = (H₁, H₂, G).**

<table>
<thead>
<tr>
<th>h₁</th>
<th>h₂</th>
<th>h₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>g₁₁</td>
<td>g₁₂</td>
<td>g₁₃</td>
</tr>
<tr>
<td>g₂₁</td>
<td>g₂₂</td>
<td>g₂₃</td>
</tr>
</tbody>
</table>

| h₁² | h₂² |

H₁ = \( \{ h₁³_i \mid i = 1, 2, 3 \} \).
\[ H_2 = \left\{ h^2_i | i = 1, 2 \right\} \]
\[ G = \left\{ g_{ij} | i = 1, 2, 3 \land j = 1, 2 \right\} \]

In this example, \( H_1 \) and \( H_2 \) are headers. The raw table skeleton is \( \{ H_1, H_2, G \} \). The Cell Connection Graph is \( \{ H_1 \rightarrow G, H_2 \rightarrow G \} \), which when added to the raw table skeleton gives the medium table skeleton. The table predicate rule (\( p_T \)) is \( H_1 \land H_2 \) and the table relation rule (\( p_R \)) is \( G \). \( p_T \) and \( p_R \), together with the medium table skeleton, gives the well-done table skeleton. Finally, the addition of a mapping that assigns a predicate expression to each \( h^N_M \) and an expression to each \( g_{NM} \) gives the full tabular expression.

In summary, the indexing scheme determines a raw table skeleton, where a table is a set of cells. The cell connection graph shows “information flow”. A table predicate rule and table relation rule provide the remaining information.

### 3.2.2.2 How does HTables define table types?

HTables (Kahl 2003) further developed Janicki’s model along algebraic lines in a functional programming setting. This required the definition of a “Table Evaluation Structure” for each new table type.

A Table Evaluation Structure (TES) is a 4-tuple of the following:

- A sequence of types, e.g. \( (T, \text{Bool}, \text{Bool}) \) for two-dimensional normal tables describing a function with range \( T \). These types are the range of the main grid and the ranges of each header grid. However, this sequence is usually not explicitly listed, but instead left implicit.

- A wrapper function, e.g. \( I \) (the identity function) is used to check the completeness of normal tables. This function is applied at the end to the formula that is derived from the other elements of the TES. Although the fact that it is merely \( I \) for normal tables might lead us to believe it superfluous, it can be used to specify different treatments of overlapping or nonexistent headers.

- A cell embedding function, e.g. \( \lambda x. \text{true} \) for normal tables. This is used to turn a cell into a table, as the defined concatenation operators all operate on tables.
A sequence of combinator pairs, e.g. \((\land, \lor), (\land, \lor)\) for normal tables. There should be one pair in this sequence for each header (or, equivalently, dimension). The first element of these pairs is the header combinator or header attachment function that is applied to the header element and the data element; and the second is the combining function that is applied across all elements in the header. The latter must be associative.

For an example of how this 4-tuple is used, consider the test for completeness of a normal table \(I\langle(\land, \lor), (\land, \lor)\rangle(\lambda x.\text{true})\). In order, these are the wrapper function, combinator pairs and cell embedding function (the sequence of types is left implicit). The cell embedding function \(\lambda x.\text{true}\) always yields the neutral element for the header attachment function \(\land\), making the header attachment function irrelevant for the purposes of this test. This means the only interesting bit is the combining function \(\lor\), that is applied to the headers to check for completeness. Finally, the wrapper function \(I\) has no effect for this test.

Obviously, each desired operation will have a different TES. I’ve only walked through testing for header completeness above.

### 3.2.3 How were expressions stored in other tools?

In the McMaster system, the kernel was responsible for the storage of tables in memory and on disk. Although the kernel did not perform evaluation, it was used by the evaluator tool. As documented by McMaster’s Software Engineering Research Group (1997) their kernel was composed of two modules:

1. Table Holder, which was the only module that depended on the representation of tables and expressions (“hid the secret” in the terminology of (Parnas 1972))
2. Information, which was the only module that depended on how symbols or variables, and the information about them, was stored

The McMaster system also had a Context Manager module outside their kernel. The term *context* was used with the same meaning as *environment* in a Lisp-like language. It is a binding of names to values. This module hid the secret of how this binding was represented.

Contexts were supported by code shared among many tools, not by the kernel. No provision was provided for nesting contexts; they all existed at the top-level. Thus, they were significantly less flexible than the static scope rules that are usual in programming languages.
HTables (Kahl 2003) is a completely separate project from the McMaster TTS. Although no tools built on it are described in the literature, a library has been written which provides similar storage functionality to the TTS kernel.

### 3.2.4 How were functions defined for reuse in other tools?

The ability to define auxiliary functions for use in specifications is indispensable when using these tools.

The McMaster TTS had a dedicated tool (the symbol editor) for defining functions that would allow them to be input and output. However, evaluation of such reusable functions required the evaluation tool code to be extended.

The NRL tools allow the easy definition of constants in one of the six dictionaries associated with a specification, but no such support is given for functions that take an argument.

### 3.2.5 How was evaluation performed by other tools?

Evaluation in the McMaster tools was the subject of Ruth Abraham’s Masters thesis (Abraham 1997), based in turn upon the Test Oracle Generator of Peters (1995). Abraham’s evaluator accomplished its task by translating tabular expressions into C code. This C code evaluated the original tabular expressions.

This approach worked well but had limitations. The operators available were initially only those supported by the C programming language, with no dedicated support for defining new operators. Using an evaluator with more built-in functions, such as a CAS or one of the many C libraries for mathematics, would have supported a richer set of conventional expressions without needing to implement them.

HTables requires that table types be defined using “Table Evaluation Structures”, which define a function to evaluate instances of the table type. This evaluation is performed by a Haskell interpreter/compiler or the Isabelle/HOL theorem prover. Thus any conventional expressions used in HTables must be evaluable in Haskell or Isabelle/HOL, depending on which is used.

For example, we can define Table 4 using the following Haskell code:

```
myTable = table1 [True, False] [1, 2]
```
Table 4. Simple one-dimensional table.

<table>
<thead>
<tr>
<th>True</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>2</td>
</tr>
</tbody>
</table>

Assume we want to check Table 4 for completeness. This can be done by defining a custom TES (derived from Kahl (2003)):

```plaintext
completenessTest = mkTES1 (mkCPR (F2 (&&)) (F2 (||))) (const True))
```

Now to test for the completeness of our table we evaluate the following:

```plaintext
tEval1 completenessTest myTable
```

NRL do not provide a general evaluator for general tabular expressions. They do however provide a “Simulator” for a system described using a restricted subset of tabular expressions, where at each event these expressions are evaluated to determine the after-state. To use this, the user loads a scenario, i.e. a sequence of monitored variable values that implicitly defines a sequence of events. Each new state is computed from a monitored event and the current state. To perform this computation, a transform function \( T \) of their requirements model is applied:

“Our model represents a system as a state machine \( \Sigma = (S, S_0, E^m, T) \) where \( S \) is the set of states, \( S_0 \subseteq S \) is the initial state set, \( E^m \) is the set of monitored events, and \( T \) is the transform describing the allowed state transitions. In our model, the transform \( T \) is a function that maps a monitored event \( e \in E^m \) and the current state \( s \in S \) to the next state \( s' \in S \); a state is a function that maps each state variable, i.e. each monitored or controlled variable ... to a type-correct value ...”

Heitmeyer, Archer et al. (2005).

An interesting point about the NRL evaluator is that it is described as an interpreter. Abraham’s (1997) evaluator was designed to generate code for other language processors to evaluate.

No provision was made by any tool for partial evaluation, where a subset of the free variables are assigned values and others are left free.
3.2.6 What else did the previous tools do?

Sections 3.2.2 to 3.2.5 describe how tasks that the TTS kernel must perform were done in other tools. However, these tools performed many other software development tasks, often using tables. Although not directly relevant to the TTS kernel, these tasks are described here for background and to hopefully clarify what requirements these tools were intended to satisfy.

3.2.6.1 What else did the McMaster tools do?

One feature that the McMaster TTS and HTables implement and we do not is table transformation (Zucker and Shen 1998). In the McMaster system this was a tool built on top of their kernel that performed table inversion, i.e. transformations between inverted and normal tables only (there are many more). To understand what this is, recall that tabular expressions are a readable representation of mathematical expressions, usually function definitions. Functions may be defined more succinctly in one table type rather than in another (e.g. inverted rather than normal), so it may make a table easier to read if we change table types while preserving the semantics. Transformation is this semantics-preserving conversion from one presentation (e.g. normal table type) to another (e.g. inverted).

3.2.6.2 What else did the SCR* tools do?

NRL’s SCR* tools are being used on military problems. Four main SCR* tools are mentioned in the literature (Heitmeyer 2002; Heitmeyer, Archer et al. 2005):

1. Specification Editors. NRL wrote a different editor for each table type that they use. They did not use a general mathematical model which would have allowed the construction of a generic editor.

2. Checkers. There are several tools to check specifications, depending on specification size, the desired level of verification and also what back-end to use to do the actual checking (SPIN (Holzmann 1997), Salsa (Bharadwaj and Sims 2000), PVS (Owre, Rushby, and Shankar 1992)). These performed some semantic checks on a specification automatically, freeing human reviewers to look for more subtle problems. There are a variety of checkers because they find different problems by using different techniques (e.g. SPIN does model checking whereas PVS proves theorems). A secondary factor in this variety is the trade-off between ease of use and capability, e.g.
TAME (Archer et al. 2002) and PVS are slow, difficult to use, and can be applied to a smaller set of problems, but may give the best results.

3. Simulator. This is described in Section 3.2.5.

4. Invariant Generator. This infers invariants, or predicates that must be true for the expression to have any meaning (e.g. for evaluation), from the specification under consideration. These can then be manipulated (e.g. proven true, false or inconsistent) using external software such as a theorem prover. Of course, since they are invariants such proofs should always be true, unless there is a contradiction in the specification or if proving the theorem is beyond the current state of the art in computer theorem proving.

As well as the tasks individual tools perform, a general feature of this system is the close integration with existing tools, currently evolving towards a file interchange format.

The SCR* tools were originally written to help in the production of requirements documents.

3.2.6.3 What did the OPGI tools do?

Ontario Power Generation Inc.’s motivation for developing a set of tools was to help in the development of nuclear reactor shutdown systems (Lawford et al. 2004; Wassyn and Lawford 2003).

A typical workflow for using these tools is the following:

1. Specification writers use an off-the-shelf word processor to write their documents and save them in Rich Text Format (RTF).

2. A set of Software Engineering Standards and Methods (SESM) tools performs basic consistency and syntax checks on these documents, delegating those checks that are semantic and that depend on the meaning of the functions to PVS.

3. These RTF documents also serve as input to the Software Design Verifier (SDV) tool.

4. The SDV tool uses PVS to check further properties of the specification, such as the completeness and consistency of table headers.
4. What tabular expressions does the kernel specification support?

4.1 Why do we need a semantics for tabular expressions?

A formal syntax and semantics of tabular expressions is useful for:

a. Ensuring agreement on how to decide whether an expression is well-formed or not
b. Ensuring agreement on what is meant by an expression
c. Defining new types of tabular expression
d. A basis for generic tool support

4.2 What were the predecessors of our definition?

Tabular expressions were first defined in Parnas (1992) after their successful informal use in the NRL SCR (A-7) project (Heninger 1980; Heninger et al. 1978). Similar informal definitions are still used by the Naval Research Laboratory (Heitmeyer, Archer et al. 2005) and Ontario Power Generation, Inc. (Lawford et al. 2004), both descended from the NRL SCR project.

After Parnas had defined a set of individual types of tables (Parnas 1992), Ryszard Janicki proposed that all were special cases of a more general formal model (Janicki 1997; Janicki 1995). This model was used in the system for support of tabular notations (TTS) developed at McMaster University (Software Engineering Research Group 1997), but the experience showed that the model could not be fully used. Several other people continued the work of placing tabular expressions on a formal foundation, including Abraham (1997); Deshernais et al. (1998); Heitmeyer, Jeffords, and Labaw (1996); Kahl (2003); Wilder (1999) amongst others.

Now a new, more general and mathematical model for tabular expressions has been developed by the Software Quality Research Laboratory (SQRL) at the University of Limerick (Balaban et al. 2007). Unlike previous work, this approach defines the meaning of a tabular expression by describing an equivalent conventional expression. The model described herein is superior to the Janicki model because:

a. The semantics are unrelated to any geometric representation
b. By defining tabular expressions in terms of an equivalent conventional expression, it is possible to take advantage of existing software that works with conventional
expressions

c. We can define many table types that could not be defined with the Janicki model, for example Tables 22 and 23 (NRL Mode Transition Tables 2 and 3 respectively) in Abraham (1997)

d. His model did not provide enough information for evaluation of all known table types

4.3 What are indexed sets?

“A function \( f \) is defined from a set \( I \) to a set \( X \). The domain of the function \( f, I \), is called the index set of \( f \), the range of the function \( f, X \), is called the indexed set of \( f \). An element of \( I \) is called an index. The triple \((I, X, f)\) is also sometimes called an indexed set.”

Halmos (1987).

4.4 What is a grid?

A grid is an indexed set, where the index set can contain anything and the indexed set contains expressions (either tabular or conventional).

The cell in the grid \( G \) indexed by an index is denoted \( G[i] \). We also use \( \{G[i_1], \ldots, G[i_n]\} \) to denote a grid \( G \) with \( \{i_1, \ldots, i_n\} \) as its index set.

\( \text{Card}(G) \) is the cardinal number of the index set for a grid \( G \). For grids this is the number of cells.

If the index set is a set of N-tuples for some fixed N then there is a natural visualisation of this indexed set as a rectangular, cuboidal or higher dimensional grid. Thus, this model of tabular expressions can represent everything that previous models could represent and more.

Note: only tuples of integers (e.g. \((x)\) for one-dimensional tables, \((x, y)\) for two-dimensional tables, etc.) were ever used as index sets for grids in the past.

4.5 What is a tabular expression?

A tabular expression, \( T \), is a triple \( \langle GS, I, f \rangle \), where \( GS \) is a set of grids, \( I \) is an indexed set, and \( f \) is a function with \( I \) as its domain and \( GS \) as its range. Briefly, a tabular expression is an indexed set of grids.
The grid in the tabular expression \((T)\) indexed by an index \(i\) is denoted \(T[i]\). We also use \(\{T[i_1], \ldots, T[i_n]\}\) to denote a tabular expression \(T\) with \(\{i_1, \ldots, i_n\}\) as its index set.

\(\text{Card}(T)\) is the cardinal number of the index set for a tabular expression \(T\). For tabular expressions this is the number of grids.

Many indexes may refer to the same set element. This means that a grid may be referred to by more than one index in a tabular expression. This is true for grids too, i.e. an expression may appear at many places in a grid.

**4.6 What are the implications of this mathematical model?**

A new mathematical model of tabular expressions is only worthwhile if it has advantages over previous models. These advantages are explained in this section.

**4.6.1 What does it permit in terms of semantics?**

This mathematical model of tabular expressions is sufficient to express all previously used tables and many others. The McMaster Software Engineering Research Group (SERG) normal, inverted, vector and general decision table types are defined in Balaban et al. (2007). Further, the table types used by the NRL and Ontario Hydro can both be defined using this mathematical model.

**4.6.2 What ramifications does this have?**

We separate the appearance of tables from their meaning. The fact that appearance is no longer a factor in how tables are interpreted allows the definition of table types that were hitherto impossible because they would have had to be defined by referring to pictures. An example is the circular tables in Balaban et al. (2007). Also, it is believed that computerised manipulation of software specifications written in tabular form will be more tractable than with previous definitions, given the well-defined semantics of the evaluation and restriction terms.

**4.6.3 How are table types defined in the new model?**

In the new model, a table type is defined by two terms, a restriction term and an evaluation term. These are either tabular or conventional expressions. If tabular expressions are used, they must be of a type that has been defined previously.
The *restriction term* defines a predicate that must be true for any tabular expressions of that type. Its domain is the set of all tabular expressions.

The *evaluation term* defines the semantics of a table type by describing an equivalent conventional expression.

Numerous examples of both of these may be found in Balaban *et al.* (2007). Some useful functions when defining table types are *Card, IndexSet, TypeG, Proper, select* and *evalG*. *Card* and *IndexSet* have already been defined. *Select* is defined as follows:

\[
select(G) = \{x | G[x]\}
\]

Informal definitions of the other functions are shown in Table 5. They are all formally defined in Balaban *et al.* (2007).

**Table 5.** Some functions used in the definition of table types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeG</td>
<td>The type of all elements in a grid</td>
</tr>
<tr>
<td>Proper</td>
<td>Predicate that tests whether a header is proper (i.e. one, and only one, cell is true)</td>
</tr>
</tbody>
</table>

Armed with these definitions, the normal table type shown in Section 1.1.2 is defined by the following restriction term:

\[
(\forall i: 1 \leq i \leq n, TypeG(T[i]) = BOOL \land \\
Proper(T[i], true))
\]

Informally, these lines mean the following:

- Each header grid contains only predicates
- The contents of header grids satisfy the disjointness and completeness requirements

The evaluation term is:

\[
T[0][select(evalG(T[1])), \ldots, select(evalG(T[n]))]
\]

which means, informally, “the cell in the main grid indexed by the Cartesian product of the indexes of which cell is true in each header grid”.
4.6.4 How does the model define other table types in practical use that aren’t defined in Balaban et al. (2007)?

Different table types have been used by many groups outside of SERG. Our mathematical model must be able to define these also. In particular, we considered the table types used by NRL and Ontario Hydro. These are defined in exactly the same fashion as the SERG table types, by evaluation and restriction terms. In Abraham (1997) it was found that all previously used table types except for two (the previously-mentioned NRL Mode Transition Tables 2 and 3) are expressible in terms of Janicki (1997). Since the new mathematical model can express all previously formalised table types, this result is true of the new model also. In addition, we can now define the table types that the Janicki model could not.

Other table types are formalised in exactly the same way as the four SERG types treated in Balaban et al. (2007), with restriction terms and evaluation terms. In particular, there is no notion of “built-in” table types. All table types have the same status.

Recall that we defined the meaning of tabular expressions on the assumption that conventional expressions are already defined to separate these two concerns. This means that we do not discuss the significant differences in the conventional expressions (e.g. modes, conditions, events and other interpretations of predicates) and document structure used by other groups.
5. What were some issues that we had to consider when designing this tool?

Some issues that we had to consider when designing the TTS kernel were as follows:

1. How can we check that function arguments are in the function’s domain?
2. How can we store and evaluate expressions?
3. How do we define functions in the function library?
4. How can we design the kernel to be resilient in the face of errors in the code that invokes it, the code that it invokes, and possible errors in its own code?
5. How do we reduce our work?

These issues are expanded on below.

5.1 What issues are discussed elsewhere in this thesis?

The more substantial problems that were encountered during the design are not described in this section at all, but have sections of their own elsewhere in the thesis. This section is a quick reference to where these may be found.

The first task was to choose a mathematical model of what the kernel will do (Section 4) and limit its scope (Section 2). These are part of the requirements-gathering activity and have already been described. Decisions made as part of the design activity are described in this section.

The question of what documents are appropriate to design the kernel is dealt with in Section 6.

How to decompose the kernel into modules according to the information-hiding principle is described in Section 7.

Conventions adopted while designing the interfaces of these modules are described in Section 8.

A way to translate the produced documentation into program code is described in Section 9.
5.2 How did we address the other issues?

5.2.1 How can we check that function arguments are in the function’s domain?

A type system is a way of classifying objects. The concept originally arose in mathematics, but lately type systems have been applied to programming languages also (Cardelli 1997).

There are two broad approaches to implementing type systems in programming languages. Both of them rely on data being somehow classified. *Dynamic type-checking* is an approach where the *data* in a programming language are classified and at run-time each function checks that it was passed an object of the correct type.

*Static type-checking* is an approach where the *variables* in a programming language are classified and at compile time it is checked that such variables are only passed to functions in whose domain they occur. Certain theorems may then be proven about a subset of the program’s run-time behaviour, usually simple, decidable theorems but not always.

A statically typed expression language seems preferable at first glance. However, the tabular and conventional expressions used in practice introduce some features that are not required in most statically-typed programming languages. Type systems such as that used in PVS (Owre and Shankar 1999) or OpenMath’s Extended Calculus of Constructions (Caprotti and Cohen 1999) would be better suited to satisfy these requirements, but these were considered too difficult to implement. In retrospect, these problems support the case made by Lamport and Paulson (1997) that most of the studied type systems are more suited to programming languages than mathematics.

Hence it was decided to use dynamic type-checking for the expression language. Under the broader definition of “classification of objects”, the specified programs in the kernel module interface check structural constraints on the data structures (e.g. that a tree node has the correct number of children). This could be considered as a kind of “typing” and indeed this is how the Maple CAS uses the term.

5.2.2 How can we store and evaluate expressions?

Programming language interpreters must also store and evaluate expressions, so a kernel to perform the same tasks has much in common with them. Designing and implementing interpreters is a mature field of computer science.
One point of similarity is that we can express the syntactic structure of the subset of tabular expressions that the kernel supports with a formal grammar. I use the notation of Crocker and Overell (2005), but the only unfamiliar syntax may be that * indicates 0-∞ instances of a token and 1* indicates 1-∞ instances of a token:

\[
\begin{align*}
table &= 1*grid \\
grid &= 1*(index \ expression) \\
index &= 1*integer^† \\
expression &= table / \ conventional_expression \\
conventional_expression &= function‡ \ *expression
\end{align*}
\]

In the TTS kernel can be seen some modules that would also be in a programming language interpreter. This inspired the choice of many of the kernel data structures, e.g. symbol tables (which we call libraries) and parse trees. The operations required on these data structures were then driven by our user requirements.

Another similarity is with a family of languages, rather than language processors. These are the array programming languages (Eriksson-Bique 2002). In these languages, the array is the central data type and most problems are solved by performing operations on arrays. The set of operations allowed on tables by the TTS kernel is not as extensive as in an array processing language. This is because our kernel will only implement primitive operations. However, it is possible to implement more advanced operations in terms of these primitives.

Finally, Prolog also uses N-ary trees as a central data type, but the radically different language paradigm means that our set of operations on trees is quite different.

### 5.2.3 How do we define functions in the function library?

Lots of programs to evaluate expressions (e.g. interpreters, compilers, computer algebra systems) have built-in functions, and allow the user to define others. However, it is also possible to have no built-in functions.

We decided that the TTS kernel should have no built-in functions, as such a design usually entails treating built-in functions differently to user-defined functions. Since built-in functions are technically unnecessary, introducing the complication of the unnecessary

---

^† In Section 4.4 we say that an index set can contain anything but that only integers have been using previously. Here we restrict the index set to integers for the design and implementation.

‡ As explained in Section 7.3, variables and constants are treated as 0-ary functions.
special case is undesirable. This decision then introduced the problem of how user-defined functions are declared and defined.

Functions are declared by supplying the following pieces of information:

1. Name
2. Domain
3. Range

Function implementations are defined by a string in the syntax of whatever evaluator is used (a computer algebra system in our initial design), with placeholders for the function arguments.

The McMaster TTS evaluator translated into C. Also, only the functions of the standard C maths library were available. This would be an adequate initial set of functions for our kernel too. This initial function library is really only for testing purposes and would need to be extended for the problem domain for real-world use of the TTS.

What may not be obvious is that in our kernel all symbols are held in the library, not just function definitions. So table types, variables and constants are stored there too. However, symbols are tagged with a “superclass”, so it is easy to get a list of only the table types etc.

5.2.4 How can we design the kernel to be resilient in the face of coding errors?

The kernel is the the lowest level that SQRL designed in the TTS uses hierarchy (Parnas 1979). Thus, the only errors that it handles are those from the support software, and even in this case all that happens is that the errors are translated to conform to the kernel interface specifications and propagated; the users have ultimate responsibility for recovering†.

Undesired events must be handled and/or reported in such a way that:

- Tools can attempt error recovery in order to continue operation
- The TTS kernel is always in a consistent state

Note: all errors are propagated to the client code—none are handled in the kernel itself.

† Replication is one such recovery method, thus whether or not the TTS runs on a distributed system is an important question for tool designers but it it irrelevant to us.
Before designing a way to handle or report errors, it is instructive to consider the different types of error that may occur. We classify errors along two dimensions as shown in the Table 6. The columns represent what other software in the system must be notified of the error. The rows represent the cause of the error.

**Table 6.** Error classification.

<table>
<thead>
<tr>
<th></th>
<th>Call Stack</th>
<th>Arbitrary Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Error</td>
<td>exc</td>
<td>Not used</td>
</tr>
<tr>
<td>Broken Mechanism</td>
<td>Not used</td>
<td>fault</td>
</tr>
</tbody>
</table>

The way in which the two error classes are handled is listed in the cell contents. Because of the two unused combinations, our error reporting policy splits errors into two classes. We define two output variables for each module, `exc` and `fault`. `Exc` is set after unexpected behaviour on the part of the caller. `Fault` is set after a failure of the invoked component. This policy could be implemented in many ways, depending on the language support for error signalling and whether it is usable. It is the responsibility of the programming language translation guide to specify how these are implemented in code.

### 5.2.5 How can we take advantage of previous work?

There is a large body of existing research on tools for manipulating tabular expressions. In particular, Software Engineering Research Group (1997) described in detail a system that had sub-components that performed similar tasks to our kernel. Studying the achievements and shortcomings of this previous work had a significant impact on our kernel requirements. However, requests by our users led to significant differences with the McMaster system.
6. What techniques did we use to document the kernel?

Parnas and Madey (1995) describe the documents needed to comprehensively document a whole end-to-end system.

We produced a module guide and module interface specifications. The module guide is written following Parnas, Clements, and Weiss (1985) with the only change being a tabular format due to Dragomiroiu. The interface specs use the Trace Function Method (Parnas and Dragomiroiu 2007) to document the relationship between the values of the output variables and the history of the values of the input and output variables.

However, the kernel is just a component in the TTS, so not all the documents are needed. In particular:

- There is no one document that forms a requirements specification. The kernel is a component in the TTS. All of the access programs in all of its modules are intended for use by the rest of the TTS. So we use our module interface specifications as a requirements specification.
- Module internal designs were not written. Given schedule constraints, we decided to follow the conventional wisdom that problems earlier in the software life cycle cost more and concentrated our efforts there.

6.1 How is the module decomposition documented?

The TTS kernel was decomposed into modules according to the principles of information hiding (Parnas 1972) and hierarchical decomposition/refinement. Information hiding is the design principle that anticipated future changes are isolated in exactly one module. Usually, every design decision is anticipated to change. The design choice is called the module’s “secret”. Choosing an alternative design option just involves replacing individual modules.

This decomposition is documented in a “Module Guide” (Parnas, Clements, and Weiss 1985) in Appendix A. A module guide is a document that lists the secret that each module hides, arranging the modules in a tree structure where the arcs represent a refinement relation, namely “part of”. That is, a non-leaf module’s secret should be equivalent to the composition of the secrets of all its children.
6.2 Why document the inter-module interfaces?

The objective of interface specifications is to accurately and concisely document the interface of the individual modules that make up the TTS kernel.

We believe that interface specification documents are a more effective means of communication than face-to-face conversation between developers implementing the module interface, testers verifying the module’s implementation and developers coding modules that use the module interface. The use of a rigorous notation resolves a lot of the ambiguities that would otherwise arise from natural language specifications or conversations. This is especially true for geographically distributed teams or when there is staff turnover.

6.3 What methods did we use to document the module interfaces, and what information did this documentation contain?

We rejected natural language as too ambiguous. Formal methods such as Z and the Vienna Development Method (VDM) were designed for specifying systems in a model-based style. In our terminology, we distinguish models from specifications by stating that the latter must be a complete description of the requirements, whereas the former is limited to some properties of interest. We want to write complete specifications, not models, so model-based methods aren’t entirely suitable. Their complexity is also very worrying. We believe that a notation using just classical mathematics, e.g. relations and recursive function definitions, is more suitable as a reference source for a reader who is not also an author.

The Trace Function Method (TFM) has two immediate ancestors. The first is the requirements method used in the SCR project (Heninger 1980). However, TFM functions are all discrete, including those on the domain of time. General time functions of the SCR model are replaced with discrete sequences representing the history of the module. Similar to requirements documents, input variables are identified for each module. However, we now also identify the output variables. The values of output variables are defined as a function (or less commonly, a relation) of the past values of input variables.

The second ancestor is the Trace Assertion Method (TAM) (Parnas and Bartussek 1977). The discrete sequences representing the history of the module are an extended version of the “traces” used in the various versions of the TAM that includes output variables as well as input variables. The defining functions are specified on the domain of a trace, or module history. A major difference from the TAM is that no canonical form for traces is
defined.

Taken together, these changes yield a method of specification using classical function definition, the same foundation that functional programming uses. We believe that this conventional foundation makes the TFM easier for professional software developers to learn than notations such as Z or VDM. This has been borne out by industrial experience to date (Baber et al. 2005; Quinn et al. 2006).

Because traces are the domain of most of our functions and our specifications are organised around output variables rather than the more conventional organisation around procedures or methods, there isn’t necessarily an obvious mapping to code (unless such code unrealistically remembers all past history). This is not the case with Z or VDM specifications, which often are models looking much like programs except for the notation.

In contrast to this work needed for coding (see Section 9), transforming a TFM specification into a test specification should be easier (Quinn 2007). For example, we can use the fact that each row in a table gives a natural partition of the input space and compare output variables against their specification when input variables are set to values to select each row in a table. More sophisticated strategies are the subject of current research.

The key concepts in TFM are events, event descriptors and traces. These are discussed below. The definitions are quoted or paraphrased from Parnas and Dragomiroiu (2007).

6.3.1 What is an event?

“A software module may be viewed as a finite state machine operating at distinct points in time, which we call events.

At each event, the module will do some combination of the following:

• read some of the global variables (e.g. via input parameters), and
• change its internal state, and
• change the value of some of the global variables.”

Parnas and Dragomiroiu (2007).
6.3.2 What is an event descriptor?

An event descriptor describes the externally-observable module state both before and after an event. The externally-observable module state is defined as the values of all input, output and input/output variables. An abbreviated event descriptor contains the same information but omits those variables that are neither read nor written during an event.

An input variable of note that occurs in every event descriptor is PGM which is the access program whose execution is associated with the event.

6.3.3 What is a trace?

A trace is a description of a sequence of events; in other words it is a sequence of event descriptors. It describes the complete history of a module.

6.3.4 What do we mean by an input or output variable?

Values that are stored where other modules can read them are called outputs. The locations where these values are stored are called “output variables”. All of the computations use values received from external components known as inputs, and their locations are called “input variables”. What appears as an output variable in one module specification may appear as an input variable in another module specification.

An input-only variable cannot be changed by the module being specified. Output-only variables can be changed by the module being specified. Input/output variables may exist, although in practice there seems to be little need for them, based on the fact that they were never used in the TTS kernel module interface specifications.

6.3.5 How do we describe the value of output variables?

The values of output variables are described using functions or (less commonly) relations on the domain of traces (input variables are not defined, just declared). These functions will typically use the basic functions, predicates and function generators from Parnas and Dragomiroiu (2007). Often, the most readable presentation of these functions is as tabular expressions due to their piecewise continuous nature. Tabular expressions can be used to define relations so many values are possible, but we usually restrict ourselves to functions in practice.
6.4 Summary

It may be helpful to describe how these documents were practically written in the case of the TTS kernel:

1. We identified the input and output variables for the TTS kernel. This fixed the kernel’s scope

2. We split our system into modules based on information hiding and documented this in the module guide, described in Section 7

3. We specified the interface of each module in the system, following the format described in Section 8

The fact that this is an iterative process (Parnas and Clements 1986) cannot be overemphasised. Initial versions of our documents bear little resemblance to their current state and indeed such revision is the essence of designing through documentation.
7. How was the TTS kernel divided into modules? Why was this particular structure chosen?

The structure of and motivation for our chosen decomposition is described in this section.

The kernel appears as a single component to other tools. The fact that it is further subdivided is hidden (Parnas 1972). A previous system (Software Engineering Research Group 1997) had a separate kernel and evaluator tool. However, in practice, changes to the evaluator frequently also required the kernel to be changed. In our TTS, evaluation is a service performed by the kernel.

7.1 What decisions are we hiding?

![Figure 1. First-level module decomposition.]

The two principal secrets of the kernel are how to (1) store expressions and (2) evaluate expressions, shown as “Storage” and “Evaluation” respectively in Figure 1. Expressions can be either tabular or conventional. The biggest secret that we hide is the fact that storage and evaluation are separated within the kernel. This is necessary because the function library is separate and no functions are built-in, but users of the kernel do not need to know this. Thus, hiding how the storage and evaluation is done allows the use of third-party software to handle either of these functions if suitable candidates can be found.

We have designed on the assumption that we will implement storage ourselves rather than source it elsewhere. The storage module is further subdivided according to the expression type. This allows us to use third-party software for the storage of one type of expression (probably conventional expressions) while storing the other with specially built software.

The interfaces treat conventional expressions as trees. However, tabular expressions require a Directed Acyclic Graph (DAG) since it is possible for one expression to be referred
to by two or more indexes in a grid, or one grid to be referred to by two or more indexes in a table. The motivation for allowing (but not forcing) this sharing of the grid and table data structures is that it allows efficient preservation of the set semantics of tables and grids. Implementing such common subexpression elimination at construction time is also possible for conventional expressions, but we hide whether or not it is done.

**Note:** there is no need to distinguish between the storage of tabular expressions and grids. This is because for this purpose we treat them identically. That is, they are just indexed sets. Expressions and grids only need to be distinguished when evaluating, since grids have no associated semantics (although it is possible for a table containing just one grid to have meaning). An argument could be made that expressions and grids need different storage representations for efficiency reasons, but treating them both as indexed sets corresponds closely to the mathematical model and is simpler.

The evaluation module implements those programs needed to perform evaluation that do not depend on the function being evaluated. The evaluation module does not hide the definitions of functions, only how to invoke these definitions. A function library, which is created by the kernel and contains programs to evaluate the functions, is also required to perform evaluation. Each function definition in the library hides the function semantics (which may be expressed as a set of ordered pairs). The evaluation module’s secret is the method used for evaluating each type of expression. There are two sub-modules, one for conventional and one for tabular expressions.

Another sub-module of the evaluation module implements the library. Several data types are stored in the library. For functions (including constants), we store the following:

- **Superclass** (defined below)
- **Domain:** this is a flat list (i.e. one that does not contain other lists as elements) of previously-defined types, disallowing higher-order functions (i.e. functions that take other functions as an argument)
- **Range:** this is also a flat list of previously-defined types, disallowing functions that return other functions
- **The code that is used to evaluate a function when it is applied**

For table types, we store the following:
• Superclass

• Restriction term (defined in Section 4)

• Evaluation term (also defined in Section 4)

For variables, we store the following:

• Superclass

  _Superclass_ is used to group symbols together for the purposes of tools using the kernel.

  A final top-level module hides the support software (OS and any non-standard programming libraries) that is used. This would be changed to port to an incompatible operating system and it should be the _only_ module that needs to be changed.

### 7.2 What decisions are we not hiding?

We decided that we would be unlikely to change the programming language used to interface to the kernel in the future, so we do not hide this design decision. We assume that it uses one or both of the currently popular imperative and object-oriented paradigms. Facilities that are provided by all widely available programming language libraries need not be hidden. For example, it is assumed that access programs, strings, lists, and files are provided by the language since such facilities are part of currently popular languages. In this case the language’s native facilities can be used. On the other hand, if we required a data structure that is not universally supported there would be a module to hide how the data structure was represented.

Our reliance on the imperative and object-oriented paradigms means that our module decomposition may very well be incorrect if a logic or functional programming language is used. We make many assumptions, among them the following:

• We are able to mutate the global state

• The dynamic call graph is a tree. In particular, we are not _forced_ to handle the possibility of the computation being partially returned to a historical state using facilities such as reified continuations (Scheme) or backtracking (Prolog)

• All objects are accessible in the same manner from the current process’s address space. That is, the runtime executes on one computing node
We have at our disposal any library facilities that are widely available in current OO/imperative languages such as C++, Java, Python, Perl and Ruby.

Some of these would not be particularly difficult to accommodate but there would be changes required to the interfaces (e.g. to pass the current state as an argument for languages without mutation).

7.3 What design decisions did we make?

Any software project will involve deciding among several arbitrary alternatives, sometimes with no clear technical advantage to any one alternative. In this case the option that minimises development effort is chosen. Such decisions are summarised here (more important decisions typically have full sections of their own elsewhere in the thesis).

- The only construct in the conventional expression language is function application.
- Applications of these functions can be represented by a tree.
- Constants are treated as arity 0 functions.
- For consistency with tables and grids, function applications are treated as indexed sets, where the indexes are argument numbers, and the indexed objects the arguments. Only the function arguments form the indexed set; the function name must be obtained through a different access program. Explicitly distinguishing a function’s name and its arguments like this is thought to be simpler than inventing a convention that the zeroth element is “special”.

7.4 How are cross-cutting concerns, such as storage, implemented in a module hierarchy?

For our purposes, separation of concerns can be treated as a synonym for information hiding. It is useful to distinguish concerns that are difficult to physically separate efficiently (called cross-cutting concerns) from those where the invocations are rare enough that this is not a problem (called core concerns).

Many abstract datatypes in the kernel need a facility for storing data between sessions. This is called persistent storage (files are a common way to implement it). This makes persistent storage a cross-cutting concern in the TTS kernel. Therefore lessons may be learned for the specification of cross-cutting concerns in future systems by considering how this function
was split among modules.

We specify such cross-cutting concerns with one interface specification, suitably parameterised. There will be a separate assignment of work for the storage of each data type. Therefore we describe several modules in the module guide that implement the interface specification, reflecting the several work packages that are needed.

![Diagram of module hierarchy]

**Figure 2.** Location of persistence modules in the module hierarchy. **Key:** arrows mean a subset relation, where both the access programs and secret of the module at the arrowhead are a subset of the access programs and secret of the module at the other end of the arrow. Many siblings are not shown on this diagram.

Figure 2 illustrates two modules, “LibraryPersist” and “ExprPersist”, that both implement the same interface for persistence. However, they are two different work assignments because although they will share some dependencies, the code to implement both modules will be quite different.

A rejected alternative is to use just one module, a “Persist” submodule of “Support System”, to perform block storage of all data types. This has the disadvantage of violating the principle of hierarchical decomposition, as such functionality isn’t an obvious part of the top-level secret “Support Software”. So the decision was made to have separate modules for persistent storage of expressions and libraries, and make these children of a higher-level module for general storage, and siblings of modules for volatile storage.

Volatile storage is treated differently and we have a separate interface for the storage of each abstract datatype. It is possible for different systems to do the volatile storage of tabular and conventional expressions, but there is no point in having separate interfaces for these two expression types for persistent storage. Persistent and conventional expressions are usually mixed and, whereas pointers or references can manage the links for volatile storage, it is rather more involved for persistent storage and we would have to invent some kind of on-disk
tree file format or convention which isn’t thought to be worthwhile.

**Note:** we only designed one implementation of persistent storage for each data type. Allowing a choice of implementations (with of course the same interface) could facilitate interoperation with other tools through file exchange, but this isn’t done at the moment.

### 7.5 How does this structure facilitate the use of previously-existing software to provide some of the services in Section 2?

Code already exists to store and evaluate conventional expressions and it is instructive to consider how this could be reused in our module hierarchy.

Volatile storage of conventional expressions could be performed by different software than that which stores tabular expressions. This third-party software would be used as the *implementation* of the module for volatile storage of conventional expressions. Note that the same *interface* can be implemented by both software developed by SQRL and software purchased from a third-party.

Conventional expression evaluation could also be done by third-party software (computer algebra systems or simpler evaluators) if this proves easier than writing our own evaluator and the expressions being evaluated are suitable (e.g. we’re highly unlikely to find an off-the-shelf evaluator for TFM functions). Whether such a system is used or not will not be obvious to users, as the chosen interface will be implemented either way.
8. What is the detailed internal structure of our module interface documents?

“We view each module as implementing one or more finite state machines, frequently called objects or variables. A description of a module interface is a black-box description of these objects.”


8.1 What sections are in all interface specifications?

A common structure for all of the TTS kernel interface specifications was found useful. Such a structure and notation to document module interfaces was previously recommended by Clements et al. (1984) but our structure is quite different. The main change is that we structure our interface specifications around the output variables, whereas the previous specifications were arranged by access program. Also, the description of access program effects is now completely formalised in TFM instead of being given in natural language. Less importantly, at a cosmetic level we no longer use punctuation conventions to indicate the purpose of an identifier (e.g. distinguishing access programs from errors).

As well as the sections listed below, there are also sections to conform to our configuration management policy: the document identifier, version and amendment history. These will not be detailed.

Each document contains the following sections:

1. System generation parameters
2. Local set definitions
3. Output variables
4. Input variables
5. Access programs
6. Trace function dictionary
7. Output variable functions

8.1.1 System generation parameters

Parameterisation allows one document to describe a family of specifications. This is often useful to reduce the length of a system specification. This section lists any such
parameters, their type and a short description of each. An example is \#ie\# from the Index interface spec, which would be replaced at implementation time with whatever concrete type is chosen for use as an element in the index, i.e. this specifies Index to be a generic type.

8.1.2 Local set definitions

This section enumerates finite sets which are used elsewhere in the specification. It is most commonly used to define the faults and exceptions that may occur during the execution of an access program. However, if any enumerated types occur elsewhere in the interface, these types are also defined here. An example is \texttt{tag\_t} which is defined in Table 7 as the set of possible tags that a node in our expression tree may have.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Type} & \textbf{Definition} \\
\hline
\texttt{tag\_t} & \{const, predconst, var, fnappl, predexpr, logicexpr, quantlogicexpr\} \\
\hline
\end{tabular}
\caption{Sample local set definition.}
\end{table}

8.1.3 Output variables

This section lists the output variables and their types. Two output variables that are used for every module are \texttt{fault} and \texttt{exc}. These signal the occurrence or non-occurrence of a fault or exception respectively, during the execution of an access program (faults and exceptions were defined in Section 5.2.4).

8.1.4 Input variables

This section has the same structure as the “Output Variables” section. An example input variable is \texttt{Oname}, which identifies which of the multiple objects managed by a module is being operated on.

8.1.5 Access programs

This section consists of a list of all externally accessible access programs, together with their parameters, and the abbreviated event descriptor for an invocation of each access program. The \texttt{exc} and \texttt{fault} output variables are omitted to save space, as they are present in most event descriptors. An example access program is \texttt{reserve}, which is described in more detail in Section 8.3.1 below.
We use the parameter names \textit{Oname} and \textit{Value} for the object name and return value respectively. \textit{Oname} is used following the multiple-object convention in Parnas and Dragomiroiu (2007). Usually, modules can be thought of as implementing a single data object, defined by the information they hide. In this convention, we instead think of a module as implementing many objects, each with the same interface. To implement this, each object must have an identifier. When the programs are used, the identifier is prepended to the operation name to identify the main operand (or message receiver, in object-oriented terminology). Any additional operands to the access program are not treated specially (this has the effect of limiting ourselves to C++/Java-style single dispatch rather than CLOS-style multiple dispatch).

There must be access programs to create and delete objects of the type. These are described in Section 8.3. Each object has a separate trace, referred to as $T_{<\text{object name}>}$. Only event descriptors in which $<\text{object name}>$ is a parameter (the message receiver is also treated as a parameter) are in $T_{<\text{object name}>}$.

The access program table could be constructed from the other information in the specification. However, this redundancy in our structure is considered useful for cross-checking during review.

\textbf{8.1.6 Trace functions dictionary}

This section defines functions that are used later to define the output variables. All functions are defined before use in order to detect circular definitions\footnote{These are not the same as \textit{recursive} definitions, which are of course supported. An example of a circular definition would be where A is defined in terms of B and B is also defined in terms of A. Syntactically there is no problem, but both functions are undefined.}. An alphabetical cross-reference is also included in this section. An example entry in this dictionary is the definition of \textit{effect}, a predicate on the domain of event descriptors that tests whether or not an event changed the internal state of the module.

\textbf{8.1.7 Output variable functions}

This section defines the values of all of the output variables listed earlier. They are defined by functions on the domain of traces. They are arranged alphabetically for ease of reference. An example function is \textit{exc}, which evaluates to the exception raised after the current event (or “none” if no exception is raised).
8.2 How was our error reporting policy specified?

As explained in Section 5.2.4, in the TTS kernel we divide errors into two categories: faults and exceptions. Occurrences of these error categories are signalled by two output variables:

- \textit{exc} is an output variable that signals the occurrence of what we call an exception, defined as an error caused by the invoker not using the module as expected (e.g. passing in invalid parameter values)

- \textit{fault} is an output variable that signals the occurrence of what we call a fault, defined as an error caused by a failure of the underlying implementation mechanism (e.g. running out of memory)

The interface documents always specify that both these variables must be set to “none” after an event if no error occurred.

8.3 How do we describe a design to make resource management more deterministic?

Popular programming languages such as C++ and Java may report memory exhaustion with a language exception\(^\dagger\). However, such an error affects every module in the system so we prefer to treat it as a fault.

A well-known problem in practice is that we would like access program invocations to appear to the calling module to be one operation, i.e. to be atomic.

One relatively simple solution to these problems is to reserve all resources \textit{en masse} at the start of an access program implementation. The only exhaustible resource the TTS kernel consumes is memory, so instead of the more conventional \texttt{new} and \texttt{delete} operators, we offer instead \texttt{reserve} and \texttt{cancelRes} access programs that reserve and release a reservation respectively on sets of objects. This interface allows a naïve implementation of the allocation of storage where \texttt{reserve} and \texttt{cancelRes} are a thin wrapper around the language’s memory allocator, but also a more complex implementation such as the one introduced by both Dijkstra and Habermann and called the Banker’s Algorithm, in which these resource requirements are used to avoid deadlock (Dijkstra 1968; Dijkstra 1977). The more complex implementation would make better use of resources, but either implementation can satisfy the specification.

\(^\dagger\) It is also common nowadays for this scenario to just cause termination of random programs (e.g. Linux’s notorious “OOM Killer”), but there is nothing we can do in this case.
8.3.1 reserve

This section describes the reserve access program. A reserve access program has the following input variables (see Section 6.3.4):

a. A set of names for the allocated objects
b. The number of objects to reserve
c. Any other parameters that affect the size of the objects. All objects allocated in one call to reserve will be of the same size, so any such parameters must be for the largest object

reserve need only perform storage allocation. Any other initialisation of the data structures must be performed by conventional setter access programs.

8.3.2 cancelRes

This section describes the cancelRes access program in the naïve implementation. A cancelRes access program has just one input variable, a set that must have been returned from a previous reserve call. If this input variable has a value which isn’t such a set, an error must be signalled.

All storage held by objects pointed to by the set is made available for reuse at some indeterminate point in the future.

8.3.3 Example

The use of these functions may be clarified with an example. Assume that we want to construct an expression tree. This tree will be composed of many nodes. These nodes are reserved at the start of the operation:

```c
ExprTree nodes[N];
ExprTree.reserve(nodes);
/* Operations to construct the tree */
```

Now, once the tree is no longer required, we can free all the nodes again:

```c
ExprTree.cancel_res(nodes);
```

Thus we see that in operation our scheme has the following differences to the conventional Java scheme:
1. Collections, rather than individual objects, are allocated and freed at a time

2. \textit{cancelRes} is specified so that the interface specifications may be implemented in languages without garbage collection (e.g. C++). However, the specification is loose enough to allow an implementation where the objects are freed “eventually” as is the case with Java

More examples of the use of \textit{reserve} and \textit{cancelRes} may be found in Chapter 10.
9. How might the TTS kernel be implemented?

This section makes recommendations to implementors†.

The interface specifications described in this thesis are not tied to the syntax associated with a particular implementation language. Therefore the interface designed for the TTS kernel is an abstraction, i.e. it can represent many real interfaces equally well. Real-world languages have many syntactic and semantic quirks, and the interface specifications do not say how to deal with these. This gap can be filled with rules for completing the interface by adding information to the abstraction. These rules will typically augment a set of code conventions chosen for the project. Examples of the latter include Pike (1989) for C, Abrahams and Myers (2001) for C++, and Sun Microsystems, Inc. (1999) for Java. Some questions that must be answered include the following:

- How are special input and output variables implemented? We define exc, fault and PGM as special variables. These may already exist in some form in the programming language’s support libraries, in which case it makes more sense to just use these implementations.

- How are ordinary input and output variables (i.e. those other than exc, fault, and PGM) implemented? Appendix B contains specification templates. That is, what look like input, output or input/output variables in the specification are in fact placeholders that may be replaced with real variables at several distinct times: when writing the specification, when programs are written or during execution. How this mapping is done depends greatly on when it is done, e.g. if we bind a placeholder to a variable at programming time, we might make use of pointers, references or memory copying, depending on the programming language facilities available. On the other hand, different techniques are appropriate to bind a placeholder to a variable at execution time.

- How are some of the system generation parameters assigned values? Note that setting a lot of these parameters (e.g. numeric limits) is not programming language-specific. However, there are also some parameters where this decision does depend on the language used, e.g. whether or not your language supports generic programming is useful information when deciding how to instantiate modules parameterised by types.

† Parts of this section are derived from documents by Simon Marr and Dennis Peters.
9.1 Which software tools (compilers, libraries, etc.) were considered and which are recommended?

Two of the most popular programming environments were considered: C++, the Boost library, and UNIX (IEEE/The Open Group 2004); and Java 1.5 SE. Java is recommended. We believe that our interface specifications are suitable for implementation in any of the currently popular programming languages. However, we do not consider the more general problem of designing interfaces suitable for any language, including uncommon ones (e.g. one using something other than the imperative or object-oriented paradigms).

If our kernel is implemented as a library written in Java, tools must then be implemented in a language that is able to use the Java calling convention, ideally without requiring any glue code.

The current module refinement presumes that a Computer Algebra System (CAS) will be used for conventional expression evaluation only. Storage of both tabular and conventional expressions will be performed by the kernel, which also takes care of converting tabular expressions into conventional expressions for evaluation. These conventional expressions will then be output to the CAS and the result read back by our kernel. The module structure isn’t incorrect if different choices are made, but certain branches (e.g. conventional expression evaluation) would need to be refined further.

9.1.1 Development environment

This section describes the recommended development environment for the TTS kernel.

<table>
<thead>
<tr>
<th>Compiler release</th>
<th>Sun javac 1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler switches</td>
<td>None</td>
</tr>
<tr>
<td>Hardware type</td>
<td>iMac G5</td>
</tr>
<tr>
<td>Operating system</td>
<td>Mac OS X 10.4</td>
</tr>
<tr>
<td>3rd party libraries</td>
<td>OpenMaple 10</td>
</tr>
</tbody>
</table>

9.2 What language constructs are suitable for implementing the concept of modules as defined in Parnas (1972) and how do they compare to each other?

Languages vary widely in their support for modular programming, from full module systems to loose conventions. Java supports classes for object-orientation and packages for
namespace management. All module interfaces must be implemented using classes.

**Note:** any convenience functions must be included as members of some class, as it is impossible to define functions outside classes in Java.

All TTS kernel definitions should be in the `ie.ul sqr1.tts` package.

System generation parameters can be implemented by generic arguments, command-line parameters, environment variables or configuration file entries, depending on requirements.

### 9.3 What rules were used to choose names in the documentation?

There are established naming conventions for most programming languages. Therefore we need to adopt a convention for forming actual access program names from the names in the abstraction. This includes conventions for common program types like getters and setters.

The Java programming standard (Sun Microsystems, Inc. 1999) recommends a particular identifier style. We also try to adhere to the naming conventions of the Java Collection interface. Complying with the established practice should make the kernel interfaces easier for experienced Java programmers to learn. Our specifications were edited to look as though we knew that Java had been chosen when they were started, even though that decision hadn’t been made then (Parnas and Clements 1986). The Java conventions we used are as follows:

1. Member names are in lower case (e.g. `reserve` in the `Index` module)

2. Class names begin with upper case (e.g. the ADT hidden by the `Index` module is implemented as `Index`)

3. Capital letters are used to separate words (e.g. `cancelRes` in the `Index` module)

4. Getters are called `getXxx` (e.g. `getLength` in the `Index` module). An exception is made for boolean getters, which begin with “is” (e.g. `isCompleted`)

5. Setters are called `setXxx` (e.g. `setElem` in the `Index` module)

6. The following conventions are adopted for similarity to Collection or `java.lang.Object` methods: an operation to get the length of a container should be called `size`, an operation to compare two objects should be called `equals`, and an operation to deep copy an object should be called `copy`
7. Access programs without any input parameters (i.e. constants) should be implemented as final static members of the class.

8. For types that we have specified in a module, access programs that return lists of this type should be implemented by methods that return iterators.

9. Some access programs convert one data type $A$ into another type $B$. These should be implemented as constructors of $B$ that take an argument of type $A$.

9.4 How could error handling and memory management be implemented?

9.4.1 Error handling

This section recommends an implementation of the mechanism described in Section 5.2.4.

Sun recommend the use of a particular error reporting technique, exceptions. However, their advice directly contradicts parts of the error reporting scheme we have chosen, for example it allows exceptions to propagate across module boundaries. This could lead to a situation where a module would raise an exception that was not documented in its interface specification.

Because of this, we recommend implementing the $exc$ output variable in our specifications using Java exceptions, but with the addition of a catch and rethrow at module boundaries. This implements the documented behaviour. Obviously, nothing needs to be done when $exc$ is “none”.

Values of $fault$ will be signalled by setting a state variable, specifically a `java.util.BitSet`. The call stack cannot be used because software that must be notified may not be in the call stack. The observer pattern (Gamma et al. 1995) was rejected because the added complexity is not needed.

Access programs should ideally be transactional in nature, i.e. in the case of an error any changes made should be rolled back. This may not be feasible for very complicated access programs, but for now none are estimated to exist in a future TTS kernel.

An example may clarify this. Table 8 shows the possible Java exceptions for the Path module.
Table 8. Path module.

<table>
<thead>
<tr>
<th>Method</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSeg()</td>
<td>java.lang.IndexOutOfBoundsException</td>
</tr>
<tr>
<td>Path(Index)</td>
<td>java.lang.IllegalArgumentException</td>
</tr>
</tbody>
</table>

According to the Path interface spec, no other methods should raise exceptions.

Note: any method that allocates memory should catch the `java.lang.OutOfMemoryError` exception and report it as a fault. Also, because Java supports object-orientation, there is no need to check for the `object_doesnt_exist` value of `exc`, as it should be impossible.

9.4.1.1 Logging

`java.util.logging.Logger`, and not standard output or error, should be used to log all messages.

9.4.2 Memory management

The implementation of memory management recommended here sacrifices efficiency for ease of implementation. However, the interface is general enough that it can easily be improved later. `reserve` and `cancelRes` can be implemented as follows:

`reserve`

Create an array of the given number of objects and initialise each using the appropriate class constructor. The scope for the constructors of kernel objects can be restricted to private in order to prevent construction using anything other than the reserve method.

`cancelRes`

Takes as a parameter an array of objects to be released. `cancelRes` sets all references to these objects to `null`. The memory used by these objects will be released by the garbage collector when it next runs.

Note: calling code that (erroneously) retains a reference to an object that has been released by `cancelRes` will prevent it from being garbage-collected.

† Even though this is “unreasonable” according to Sun’s documentation.
9.4.3 Concurrency

No low-level threading policy is specified by the module interface specifications. However, it is imperative that the specified behaviour be fully implemented when more than one thread exists.

Because of this, a kernel implementation must lock its internal data structures on behalf of its clients. Making all publicly-accessible objects synchronized will do this.
10. How can the services offered by the TTS kernel (as described in Section 2) be used by client software?

This section illustrates how TTS kernel features may be used by tools written in Java.

10.1 How can errors be handled?

Before considering the technical details, the requirements in the case of errors need to be teased out. For tools that are implemented as libraries that themselves make use of the kernel, errors may be either handled or propagated. Errors that are propagated must be translated to errors that are in the documented interface of the tool module. For a non-interactive tool that performs only one task, we may just want to abort upon an error. On the other hand, for tools that are implemented as a long-lived GUI process we do not want to lose the user’s work, e.g. by throwing an exception to be handled by a top-level handler that offers to save the user’s work then exits gracefully.

Any call to an access program that is documented as possibly changing the fault variable must be followed by a check of that variable. Consider the following example:

```java
Library.reserve(libs); /* Documented as potentially setting fault */
if (!Library.fault.isEmpty()) {
    saveWork();
    System.exit(1);
}
```

In this case we choose to abort the program, which is often the correct choice for memory exhaustion.

As for the exc variable, which according to Section 9 we implement using Java exceptions, there are several possible strategies depending on the requirements.

For a non-interactive tool that performs only one task, not handling the exception will propagate it to the top level and terminate the process.

For tools that are implemented as libraries that themselves make use of the kernel, errors that are propagated must be translated to errors that are in the documented interface of the tool module. Consider the following snippet of code from such a tool:

```java
try {
    Sym.cancelRes(libSyms); /* Documented as potentially setting exc */
} catch (SymsUsed e) {
```
throw new ToolError();
}

Here we catch all exceptions from the kernel and translate them to the documented interface of the tool’s module. The ToolError exception is assumed to be part of this interface.

For GUI programs, we recommend handling the exc variable by either recovering or throwing a new exception to be handled by a top-level handler that offers to save the user’s work then exits gracefully.

10.2 How do we begin a transaction?

It is a requirement that it be possible to write tools in such a way that each operation will either succeed or have no permanent effect on the kernel. This is akin to transactions in databases. We help tool writers to do this by forcing all resources to be reserved ahead of time. All resources (memory is the only resource that the TTS kernel uses) are reserved at the start of the transaction. If any of these reservations fail, the transaction is aborted. At this point, no changes have been made, so there is no rollback to be done:

```java
Library libs[1];
Sym libSyms[4];
Sym exprSyms[2];
Library.reserve(libs);
Sym.reserve(libSyms);
Sym.reserve(exprSyms);
```

We must then check whether or not these memory allocations succeeded. In the case of failure, we assume that the requirement is to abort the program:

```java
if(!Library.fault.isEmpty() || !Sym.fault.isEmpty())
    System.exit(1);
```

10.3 How do we end a transaction?

The only thing that needs to be done to end transactions is to free any used resources. We do this using the cancelRes method. For example, if all resources are referenced by the arrays libs, libSyms and exprSyms, we free them all using the following:

```java
Library.cancelRes(libs);
Sym.cancelRes(libSyms);
Sym.cancelRes(exprSyms);
```
10.4 How do we define symbols in the library?

An initial library must be set up for use by subsequently-created expressions. There are two possible ways to set up such a library: making access program calls to create the data structures or restoring a library that was previously created in this way from disk. Usually, the initial library will be loaded from a file on disk as follows:

```java
lib = Library.load("the_library.dat");
```

From then on, symbols can be incrementally added to this library using method invocations.

We illustrate how the library functionality can be used with snippets of Java code. If a real test case was made out of such snippets, and the decision made to treat this test case as a transaction, memory for the data structures (library and symbols) would have to be reserved ahead of time. We assume that this has been done.

When initialising each symbol the “superclass” attribute must be set. Superclasses in our TTS kernel library are nothing more than conveniences for tool writers to group symbols (e.g. for presentation in a user interface). Note also that the initial library is not completely empty but contains a SUPERCLASS symbol, as we must have a parent for superclasses that we create. With this in mind, we can create a superclass for all functions:

```java
libSyms[FUNCTIONSYM].setSuperclass(libs[0].getSym("SUPERCLASS"));
libs[0].addSym("function", libSyms[FUNCTIONSYM]);
```

To declare the superclass of all floating-point numbers (not real, as we have to evaluate on a computer):

```java
libSyms[FLOATSYM].setSuperclass(libs[0].getSym("SUPERCLASS"));
libs[0].addSym("float", libSyms[FLOATSYM]);
```

Armed with the above two declarations, we can create a symbol to perform addition:

```java
libSyms[PLUSSYM].setSuperclass(libs[0].getSym("function"));
libs[0].addSym("+", libSyms[PLUSSYM]);
```

Because this is a function, we must specify its domain and range. These, in turn, are just lists of previously defined symbols. To declare the domain \( \text{float} \times \text{float} \):

```java
ArrayList plusDom = new ArrayList();
plusDom.add(libs[0].getSym("float"));
plusDom.add(libs[0].getSym("float"));
libSyms[PLUSSYM].setDomain(plusDom);
```
To declare the range \texttt{float}:

```java
ArrayList plusRng = new ArrayList();
plusRng.add(libs[0].getSym("float");
libSyms[PLUSSYM].setRange(plusRng);
```

We define the implementation as a string in Maple syntax, with some placeholders. This is acceptable, as the library is evaluator-specific. So if we want to use a different evaluator back-end, another library would be written with implementations in a different syntax:

```java
libSyms[PLUSSYM].setImpl("((\$1)+(\$2))");
```

A superclass for all variables would also be required:

```java
libSyms[VARIABLESYM].setSuperclass(libs[0].getSym("SUPERCLASS");
```

```java
libs[0].addSym("variable", libSyms[VARIABLESYM]);
```

Constants are represented as functions that take no arguments. So the associated symbol will have an empty domain, a \texttt{float} range and an implementation that the evaluator will understand as the constant’s value. A symbol for this constant is added to the library with \texttt{addSym}, similar to above:

```java
exprSyms[TWOSYM].setSuperclass(libs[0].getSym("function");
exprSyms[TWOSYM].setDomain(new ArrayList());
ArrayList twoRng = new ArrayList;
twoRng.add(libs[0].getSym("float");
exprSyms[TWOSYM].setRange(twoRng);
exprSyms[TWOSYM].setImpl("(2)";
```

```java
libs[0].addSym("2", exprSyms[TWOSYM]);
```

A library can be initially constructed and extended with method calls as in this section, then saved to disk for future use.

There are access programs for getting lists of symbols that have been grouped for the convenience of tools. For example, if a tool requires a list of all functions defined in the current library, it can query for the superclass “function”. Likewise, “variable” and “table” could be used to get a list of all variables and table types, respectively.

\textbf{10.5 How is a conventional expression stored?}

Consider the case where a tool needs to create an expression tree to represent “\(x + 2\)”, perhaps for later evaluation. We assume that an initial library has already been created so that we can use references to the symbol + in our expression tree. We also assume that symbols
have been created for \( x \) and \( 2 \) as shown in Section 10.4.

Now we can create the root of the tree that contains the “+” symbol. We assume that \( \text{nodes[PLUSNODE]} \), \( \text{nodes[XNODE]} \) and \( \text{nodes[TWONODE]} \) have already been reserved:

\[
\text{nodes[PLUSNODE].setSym(libs[0].getSym("+"));}
\]

Next, the child nodes are set to the symbols \( x \) and \( 2 \):

\[
\text{nodes[XNODE].setSym(libs[0].getSym("x"));}
\]
\[
\text{nodes[TWONODE].setSym(libs[0].getSym("2"));}
\]

Now when we want to add these children to the expression tree rooted at \( \text{nodes[PLUSNODE]} \), we need to specify where they should be added. This information is represented by a path, which in turn is made up of indexes. We require two paths of length one (called \( \text{paths[XPATH]} \) and \( \text{paths[TWOPATH]} \)), the former containing an index to select the first argument of + and the latter containing an index to select the second argument:

\[
\text{indexes[0].setElem(0, 0);} \\
\text{indexes[1].setElem(0, 1);} \\
\text{paths[XPATH].indexToPath(indexes[0]);} \\
\text{paths[TWOPATH].indexToPath(indexes[1]);}
\]

Now we can construct the links between nodes in the tree:

\[
\text{nodes[PLUSNODE].setSubexpr(paths[XPATH], nodes[XNODE]);} \\
\text{nodes[PLUSNODE].setSubexpr(paths[TWOPATH], nodes[TWONODE]);}
\]

This completes the construction of all kernel data structures needed to store the expression “\( x + 2 \)”. The fact that it took quite a few access program calls is not a cause for concern, as these calls will be made by other programs, not people.

### 10.6 How are expressions evaluated?

An expression such as “\( 2 + 2 \)” does not require any information other than what is already in the library to be evaluated. However, to evaluate an expression such as “\( x + 2 \)”, we must provide an assignment to set a value for \( x \), say 3. No such assignment is required for the symbol 2 because its value is part of its library definition. We assume that a constant, \( xval \), has been added to the library as shown in Section 10.4 with the implementation “\((3)\)”.

Now we create a degenerate expression tree, rooted at \( \text{valnodes[0]} \), with just one node, an application of the function \( xval \):
valNodes[0].setSym(libs[0].getSym("xval"));

Finally, this expression is added to the assignment assigns[0] as the value for the variable x:

assigns[0].setVal(libs[0].getSym("x"), libs[0].getSym("xval"));

Once the assignment has been constructed, evaluation itself is relatively simple:

rslts[0].eval(nodes[PLUSNODE], assigns[0], true);

where rslts[0] is a previously reserved expression to hold the result.

10.7 How can the tasks of storage and evaluation be performed for tabular expressions?

The interface for tabular expressions is the same as that for conventional expressions, except for the following changes:

- Paths to identify elements in a conventional expression are composed of indexes that only ever have one element (the child number in the expression tree), whereas paths to identify elements in grids may be composed of indexes with more than one element if the grid has more than one dimension

- Grids have no meaning in isolation, so can’t be evaluated. The apparent special case where a table has only one grid is not, in fact, special because even though it looks like a grid when printed on paper, this grid is in fact semantically nested within a table (giving the usual evaluation and restriction terms)

As in Section 10.5, we demonstrate the kernel functionality with snippets of code, in this case part of the code needed to construct a tabular expression. For our example we choose a two-dimensional table, $T = \{G_0, G_1, G_2\}$.

The first step is to create the index sets (sometimes called shapes) for tables and grids. Our example requires three one-dimensional index sets, one each for grids 1 and 2 and one for the table itself. These are assumed to already have been reserved, at the start of whatever transaction we’re doing. In our kernel interface, index sets are defined by an index which is one beyond the maximum in every dimension:

1dshapes[TBLSHAPE].setElem(0, 3);
1dshapes[SHAPE1].setElem(0, 2);
1dshapes[SHAPE2].setElem(0, 3);
This sets $IndexSet(T) = \{0, 1, 2\}$, $IndexSet(G_1) = \{0, 1\}$, and $IndexSet(G_2) = \{0, 1, 2\}$.

One two-dimensional index set is required, for the main grid (grid 0):

```c++
2dshapes[0].setElem(0, 2);
2dshapes[0].setElem(1, 3);
```

This sets $IndexSet(G_0) = \{0, 1\} \times \{0, 1, 2\}$, where $\times$ is the Cartesian product.

These index sets must be initialised before we can create the required grids and table, as they are a parameter to the `reserve` method for these data types.

From now on, the interface is identical to that for conventional expressions. For example, consider how to set cell (1, 1) of grid 0 to a conventional expression “$x + 2$” constructed in a manner similar to that described in Section 10.5. We identify cell (1, 1) of grid 0 by means of a path that first selects grid 0 and then cell (1, 1) within it. This requires the initialisation of two indexes, one for each step:

```c++
grid0Idxs[0].setElem(0, 0);
cell1_1Idxs.setElem(0, 1);
cell1_1Idxs.setElem(1, 1);
```

Now these indexes are concatenated into a path. Note that only concatenation of paths is supported, so individual indexes must be converted to paths of length one beforehand. The objective is to construct the data structure shown in Figure 3.

```c++
tblPaths[TMP].indexToPath(grid0Idxs[0]);
tblPaths[CELLPATH].setSeg(0, 0, tblPaths[TMP]);
tblPaths[TMP].indexToPath(cell1_1Idxs[0]);
len = tblPaths[CELLPATH].size();
tblPaths[CELLPATH].setSeg(len, len, tmp);
```

Finally, now that we have identified the cell of interest, we can change its value:
Evaluation is performed identically in the case of tabular expressions. This is a reflection of the mathematical model, which makes no semantic distinction between conventional and tabular expressions, merely a syntactic one.
11. Conclusion

11.1 What was learned?

As well as the positive points of rigorous documentation, this project also exposed some of their shortcomings. Although it has been well-known for a long time, it is worth remembering that human reviewers no longer see mistakes in a document with which they are familiar, so review is less effective than you would imagine. A way to mitigate against this would be to simulate the specifications as well as review them (and such a project is currently underway at SQRL).

This may be an indication that there is less separating specification and prototyping than is usually thought, if both activities are done correctly. Prototyping should be as high-level as possible for a quick turnaround time and specification should be as formal as possible to answer questions as early as possible. A notation based on mathematics satisfies these requirements. The gap between Prolog and the TFM is not that great.

Although it was not new, this research also confirmed that the design of interfaces composed of access programs through mathematically rigorous documentation is effective. The successful application of trace-based tabular documentation outside its traditional real-time or embedded domains, in our case for Computer Aided Software Engineering (CASE) tools is also an interesting anecdote.

11.1.1 What was learned about specifying systems?

There are two approaches to writing specifications: depending on a particular implementation language or trying not to have such a dependency. This work tried to keep the specifications relatively language-independent and so accrued many benefits, among them that the design does not overly rely on particular language features that may be badly supported by other languages. This allows reuse of requirements documents and designs, even where the code can’t be reused. There is a price in designing for the common feature set, but this was found to be acceptable. Indeed, although it is outside the scope of academic research, the exclusion of uncommon design concepts may make sound business sense.
11.1.2 What was learned about specifying systems using the TFM?

The TFM, like its competitors, is intended for the specification of large systems, so traditional case studies such as stacks do not tell us much. Although the TTS kernel is certainly not a large system by any stretch of the imagination, initial impressions are that the complexity of TFM module interface specifications scales linearly with the number of modules. A lot of credit for this is due to the use of conventions for things like the specification structure, resource management and error handling. Such a set of project or organisation-wide policies should be instituted before any use of the TFM, whether along the lines of Section 9 for Java or otherwise for different programming languages.

It is important to realise that the specification techniques described in this thesis do not replace existing best practice, but augment them. For example, test plans still need to be written and executed, although now they can be derived from the behaviour specifications. Prototypes (Robertson and Agustí 1999) should still be constructed, although now they are used to make decisions in the specification documents. Similar arguments can be made for all of the traditional software lifecycle deliverables, that they are still required but that rigorous specifications may now help write them.

11.1.3 What structural conventions for software documentation were discovered?

One of the contributions of this thesis is the common structure for module interface specifications described in section 8.1. However, there are some smaller recommendations also.

Ideally, statements should only need to be made once. However, people rarely apply this principle to the important case where something is stated both informally and formally. We consider the formal statement preferable and would like to avoid providing the informal statement for the final review copy.

This also motivated our use of a central lexicon for the specifications, where functions (usually on the domain of traces) are defined once and referred to from the specifications themselves. Importantly for impact analysis, this relationship was bidirectional and the lexicon also has a record of where each function was used. This is not new but is more reminiscent of Library or Information Science than Software Engineering, echoing the recommendation that Chief Programmer Teams include a librarian.
11.1.4 What was learned about applying the SCR techniques to the domain of symbolic processing rather than the more usual real-time/embedded control?

Predecessors of the software development method used in this thesis have seen most application in real-time and embedded domains. The TTS kernel project shows that it is equally applicable to the domain of CASE tools. However, this involved some simplifications and some complications:

• The system is wholly digital, so there is no need to consider the measurement error in analogue-to-digital conversion

• Vendors may trade off the reliability of desktop operating systems in favour of cost or features. However, with suitable conventions for performing tasks where the OS interfaces didn’t allow us to write deterministic specifications (e.g. resource allocation) this can be mitigated against

• Because the TTS kernel is a program library, we use the module interface specifications as our requirements

Given the TFM view of computers as state machines where the before and after states are specified using a relation, case studies in business software might yield some interesting synergies, as the relational model is applied differently there.

11.2 What improvements to the method and extensions to the software are worthy of further investigation?

As with most research, attempting to answer the questions in this thesis raised many more.

11.2.1 How can documentation be improved?

The cost of developing software according to the method used in this thesis might be reduced by some further investigation.

Although it is of limited use in writing specifications, visualisation has been found useful for reviewing them (Dulac et al. 2002). Exploration of similar visualisations of a software system documented using tables and TFM may reduce the cost of review. Ideally, the effectiveness of such techniques would be measured experimentally.
This thesis tried to develop as robust a system as possible, for example using a memory reservation policy. In an industrial setting, a cost-benefit analysis may recommend scaling back such measures in favour of whatever mechanisms are already provided, even if they are error-prone (e.g. overcommitting virtual memory in contemporary UNIX kernels).

Our interface specifications are language-independent but not paradigm-independent. The Larch project (Guttag et al. 1993) also considered the issue of language-independence and recommended that portable specifications be written first, then language-specific ones. However, this injunction must be weighed against the fact that it proves very difficult to convince people to write one set of documents, never mind two.

11.2.2 Briefly, are there any complementary approaches to improving software quality?

Quite a lot of work has been done on annotating source code with logical axioms about the software (Chalin et al. 2006; Leavens 1996; Sannella 1991). The extent to which such embedded documentation (e.g. annotations) complement and can replace external documentation is an interesting question.

If the fundamental computer architecture changes to massively parallel or distributed, quite a lot of this thesis may need to be revised. Current programming languages put the responsibility for properly utilising massively parallel hardware on the programmer’s shoulders. If this proves beyond most programmer’s abilities, a shift from the imperative/object-oriented paradigm to logic or functional programming may allow for automatic parallelisation, but this is far from clear at this point in time. If such a hypothetical future programming language violates the assumptions in Section 7.2, significant adaptation, verging on a redesign, might be needed.

Our design tries to help with error handling through mechanisms such as our error reporting policy and reserve. If in future distributed systems become more popular, a fail-fast architecture with replication and supervision hierarchies (Armstrong 2003) may be simpler than one that tries to handle errors. (Armstrong 2003) also argues that designing network protocols rather than interfaces based on access programs is a preferable alternative to aspect-oriented development, as cross-cutting concerns (such as fault tolerance) can then be implemented by interposing protocol translators, relays, multiplexers and demultiplexers, instead of requiring a weaving tool.
Defining an interchange file format is currently a popular way to achieve system interoperability. The alternative of defining an interface composed of access programs instead is less popular and is usually only done after many years (e.g. the 20 years from the UNIX OS’s 1969 birth to its 1989 partial standardisation by POSIX). I argue that this need not be the case and that interfaces composed of access programs can be designed through documentation just as easily as interchange formats.

It is important to remember that some research areas with “modular” in their name have quite a different emphasis to this research. In particular, modular programming as researched by Niklaus Wirth, and even arguably the OMG’s CORBA technology, is concerned with modules from analysis, through design and on into the final code. We restrict ourselves to modules in the design, which may or may not be implemented as modules in the code depending on the capabilities of the programming environment.

11.3 How would we change the design of the kernel?

The current kernel design is believed largely satisfactory for the chosen platform (Java on a desktop computer) and ones similar to it (e.g. C++/Boost/UNIX). One of the few considered changes would be the central role of a CAS in our design. It was originally motivated by the fact that tabular expressions are a presentation of conventional expressions, and CASs are a popular class of tool to work with conventional expressions. However, in practice, the conventional expressions used in our specifications made heavy use of discrete trace, rather than continuous general mathematical (e.g. $\sin$, $\cos$), functions. This emphasis on discrete rather than continuous functions means that a CAS is not really suitable for evaluating TFM expressions. And if we’re writing an evaluator anyway, the non-trace mathematical expressions used were actually quite simple, so might be more easily evaluated by us instead of trying to comply with a vendor’s interface.

11.4 How would we change how the kernel was designed?

This is quite a different question to the previous one. Here we consider how we would develop the kernel differently were we to do it again, rather than the end product. Of course, these suggestions are a product of the knowledge gained during the TTS kernel design process.
Probably the biggest change based on what was learned during this project is to pick more helpful technology to build upon and construct a complete prototype (prototype code was written, but only to answer narrowly-focused questions, e.g. about the software that we depended on). For example, the following might have been done after the initial draft of the module interface specifications:

- Choose a language more suitable for prototyping. Requirements for such a language would include code/data equivalence, including a reader and no compile/run-time split. This would allow us to extend the language’s native evaluator to handle tabular expressions instead of implementing our own evaluator. This saves the effort of implementing the symbol tables and expression trees that make up most of the kernel modules. Suitable languages might include Prolog and Scheme. There are better alternatives, such as PVS or Common Lisp, but they have a much steeper learning curve.

- Choose a markup language as a document format. This is no help at all for implementing the kernel, but an input and output tool prototype could be constructed as a preprocessor, allowing us to use the prototype to write the kernel interface specifications. Precedents for such a preprocessor in the *troff* system (Dougherty and O’Reilly 1987) include *grap* (for drawing graphs) and *chem* (for drawing molecules), which are preprocessors for the *pic* drawing package, although we would require a preprocessor for table layout instead. *Troff* may be considered old technology nowadays but a very similar design could be implemented using HTML and MathML.

- The initial library need only support the tabular expression types actually used in the kernel module interface specifications: these are one and two-dimensional normal, inverted and circular tables. However, it should still be possible to extend this library in the future.

- Initially support only the conventional expression types actually used in the kernel module interface specifications: these are the basic arithmetic and relational operators (even less than the C library) and the trace functions from Parnas and Dragomiroiu (2007).

This prototype could then have informed decisions that were taken when writing the formal specification. It would have allowed us to “eat our own dog food”, in the sense of using tools as well as using the method.
12. References


Archer, M., Heitmeyer, C. L., and Riccobene, E., “Proving Invariants of I/O Automata with

Armstrong, J., Making reliable distributed systems in the presence of software errors, The

Methods of Software Specification: a Case Study” in Proceedings of the International


Bharadwaj, R. and Sims, S., “Salsa: Combining Constraint Solvers with BDDs for Automatic
Invariant Checking” in Proceedings of Tools and Algorithms for the Construction and

Caprotti, O. and Cohen, A. M., A Type System for OpenMath, The OpenMath Consortium
(1999).

Cardelli, L., “Type Systems” in Handbook of Computer Science and Engineering, CRC Press
(1997).

Carette, J., “Understanding Expression Simplification” in Proceedings of the International
Symposium on Symbolic and Algebraic Computation (ISSAC 2004), Santander, Spain
(2004).

Chalin, P., Kiniry, J., Leavens, G. T., and Poll, E., “Beyond Assertions: Advanced Specifi-
cation and Verification with JML and ESC/Java2” in Proceedings of Formal Methods for

zation for Specifying Abstract Interfaces,” Tech. Rep. 8815, Naval Res. Lab., Washing-
ton, DC, USA (1984).


Heitmeyer, C. L., Archer, M., Bharadwaj, R., and Jeffords, R., “Tools for constructing requirements specifications: The SCR toolset at the age of ten;” *International Journal*


Chapter 8.


Quinn, C., *Quality Assessment in Real-time Software*, University of Limerick, Ireland (12 Jun. 2007).


