20

TTT — A simple type checked C language abstract data type generator

W.J. Toetenel
Faculty of Information Technology and Systems
Section Software Engineering and Programming Languages & Compilers
Delft University of Technology, Zuidplantsoen 4, 2628 BZ Delft, The Netherlands, tel. +31-152782518, fax. +31-152787141
E-mail: w.j.toetenel@twi.tudelft.nl

Abstract

The paper addresses a simple abstract data type facility for C programming, based on the VDM-SL specification language. Its main design goals were simplicity, concise representation and fast access of its data type values. The tool is under development. Test versions are succesfully applied within the context of compiler construction education and a large tool construction project.

Keywords

abstract data type facility, C-language, VDM-SL

1 INTRODUCTION

The C [8] language has been used widely since its perception in the early seventies as a system implementation language. The evolution of concepts related to data structures such as data encapsulation and information hiding led to the development of the object-oriented paradigm and resulted in the C++ [14] language as the new system implementation language in the eighties. The evolution of distributed computer systems and the spectacular growth of technical capabilities of the communication infrastructure finally led to the recent development of the Java [1] language with the Internet as execution platform. The language evolution starting from C through C++ into Java has brought many new language features into the reach of system programmers. But the basic data types like integers, characters, reals, arrays and records (objects) are preserved in the newer languages and form the basic builting elements for complex data structures. More abstract data structuring mechanisms, based on mathematical models such as sets, sequences, maps, cartesian products, composite and union types remain largely within the context of system spec-

Systems Implementation 2000 R. Nigel Horspool (Ed.)

ification notations like Z [13] and VDM [6]. Early recognition of the need for more abstract data structuring mechanisms in programming notations has led basically to two different approaches to bridge this abstraction gap, (i) the development of more abstract programming notations, such as CLU [10], SETL [12] etc. and (ii) the development of abstract data type interfaces for C and C++. The latter development has been exercised widely within the field of compiler technology where the need for abstract interfaces evolved naturally from the phased structure of modern compilers. A typical compiler consists of several highly independent processes that share abstract views of program representations, generally constructed as complex tree and graph forms. Compiler toolkits like Eli [3] and Cocktail [5] offer the means to manipulate these abstract program representations, including constructors, readers, writers and transformers. As these toolkits are targeted to compiler construction they support in particular tree (and graph) abstractions. The tool presented in this paper extends this functionality by adding other abstractions as well. As such the application area is widened.

TTT is a simple tool to generate C-language data structures for multi process applications . It offers a data type definition facility similar to the VDM-SL data definition sub-language [2], and a data generation facility that offers a complete range of data manipulation tools for creation, manipulation and destruction of data type values. The overall style of the generated interface is similar to the interfaces generated by tools such as AST [4]* and IDL [11]. The main difference between TTT and AST is the abstract data type interface in TTT, which is lacking in AST. The main difference between TTT and IDL is the simplicity of TTT's data definition language (DDL) and the dynamic type-checking facility of the constructed data type values in TTT. In IDL all data types are static, and as such their structure is completely known at generation time. DDL built-in data types include composite and union types, set, sequence and map types, tuple types and basic types such as integer, boolean and real types. Further DDL enables the definition of complex user-defined data types based on these built-in data types. For each built-in data type TTT offers a complete range of value manipulating functions, including the support for composite type values, set, sequence, map and tuple values. The support for complete tree and graph structured data values includes type checking facilities, copying, relational operators (equality and inequality), fast reading and writing operations, traversal operations, destruction and memory management operations and simple property management operations.

The usage of TTT is simple. The first step is the development of a DDL description of the data types for which support is to be generated. Next a single application of the TTT generator results in a pre-compiled library containing all generated support functions and a C language header file which embodies the interface to the library functions. Now the user can develop applications

*AST is part of the Cocktail toolbox.

Table 1 TTT type operators

name	example	values
optional type	`[ T ]`	values belonging to `T` ∪ `NIL`
set type	`T set`	set of `T` values
seq type	`T seq`	sequences of `T` values
map type	`DT -> RT`	sets of relation maplets (tuples), registering a mapping from a domain type value to a range type value
product type	`TL * TR`	creates tuples of values, binding together values of the two operand types
union type	`T1 | T2`	holding values of `T1` ∪ `T2`

using the library functions of the generated library. The library is created by the GNU C language translator tools.

This paper highlights in particular the DDL facility, the generated C language binding and the approach taken for dynamic type checking. The structure of the remaining paper is as follows. Section 2 presents shortly the DDL facility of TTT. Section 3 summarizes the generated C language binding. Section 4 presents some results of benchmarking and relates the results to the performance results of the AST tool. Finally section 5 projects the current capabilities of TTT into the future.

2 DATA DEFINITION LANGUAGE

A TTT DDL specification consists of a series of definitions. A definition is either a user defined type definition or a composite type definition. Composite types are called records.

Definitions Definitions are either type definitions or record definitions. The main difference between types and records is the set of applicable type operators. Type values are constructed by application of the TTT type operators (see in figure 1). Record values are constructed by a record constructor. For each record type there is a specific record value constructor. In this paper often the term *node* will be used to denote record values. A TTT generated data structure consists of interconnected nodes. It is not neccesary a tree in the mathematical sense, but actually a graph structure. However, in many cases it is convenient to view the data structure as a tree, consisting of a top node (root) and its descendents. We will use the term *tree* to denote a interconnected structure of nodes from which a specific node is assumed to be the top node.

Type definitions Type definitions can be used to create named type expressions. The defined types serve as reference for type checking. E.g. in

ddl 1

```
integerset = integer set;
```

a new type is defined, named `integerset` which describes values consisting of sets of integers.

Record definitions Record definitions define the structure of node values. Each node value is associated to either a user defined record definition or a built-in pre-defined record definition. The type definitions can be applied to create record values.

ddl 2

```
integersetrecord ::
    setvalue : integerset,
    card     : cint;
```

The record definition in ddl example 2 defines a record structure consisting of two fields, `setvalue` which may hold an integer set value and `card` which may hold an integer value. Each field is characterized by its field name. The values that may be contained in the field are defined by the type expression following the field name, such as `integerset`, or `ctype`, which relates to the native types from the C language binding.

Definitions may be recursive. The recursion must be guarded by appropriate type operators to get meaningful definitions.

ddl 3

```
rectype = ([rectype]) * integer;
```

In example 3 the recursion in the type definition is guarded by the optional type operator `[ ... ]`. The type `rectype` defines recursive tuple data values like `(((NIL, 3), 2), 1)`, where `NIL` is a special value, denoting the polymorphic value null.

ddl 4

```
r1 :: f1 : r2,
      f2 : integer;
r2 :: f1: r1,
      f2: integer;
```

In specification 4 a mutual recursive pair of record definitions is presented that does not have any useful semantic interpretation. The semantic interpretation (the denotation) of recursive definitions is a (possible infinite) set

of definitions. To assert that recursive definitions have a valid denotation, TTT applies an algorithm that computes the base of the recursion, if present. The base of the recursion is the smallest possible set of non-recursive type definitions having a valid denotation.

Type expressions Type expressions are created by combinations of type operands and type operators. The TTT DDL notation offers standard abstract data type operators like `set`, which creates finite set values, `seq`, which creates finite sequence values, `->`, which creates finite map values, and `*`, which creates product values. The TTT DDL offers operators to construct optional types, using the optional type operator `[ ]` and union types, by using the union type operator `|`. The optional type consists simply of a type. It is interpreted as a shorthand for `type` ∪ { `NIL` }. The union type operator is interpreted as the normal set-theoretic union.

The base type operands in DDL are type names and the DDL basic types `string`, `token`, `integer`, `long`, `byte`, `char`, `float`, and `double`.

Type names correspond to the associated definitions (either type definitions or record definitions). In example 2 the type name `integerset` corresponds to the type expression defined in 1.

In TTT the basic types are pre-defined record definitions. The values defined by the basic types are record values that hold values of the associated basic types in the C language binding. Next to these pre-defined record definitions, TTT offers the inclusion of C language basic types in record field definitions. These types are denoted by the `ctype` type, including `cstring`, `ctoken`, `cinteger`, `clong`, `cbyte`, `cchar`, `cfloat`, and `cdouble`. These types denote flat C language values belonging to the corresponding C language types `char *`, `token`, `int`, `long`, `byte`, `char`, `float`, and `double`.*

Basic types and ctypes Each basic type corresponds to a pre-defined DDL record definition (see figure 2). A value belonging to a basic type is thus a record value. This is in contrast to the values belonging to the ctype class, which are ordinary C language values (flat values).

ddl 5

```
node :: intnode  : integer,
        intvalue : cint ;
```

In specification 5 a record is defined with two fields, `intnode` which will hold record values and `intvalue` which will hold C language `int` values. The usage of ctype values are restricted to record fields. They realize simple node attributes. Ctype values may not be used within type expressions. For example when we want to model a record field as a product value of two integer values, the basic type `integer` should be used, like in

*The C type `token` is defined as `int`

Table 2 Predefined basic record definitions

DDL type class	Predefined Record Definition
`string`	`STRING :: sval : cstring;`
`token`	`TOKEN :: tval : ctoken;`
`integer`	`INTEGER :: ival : cint;`
`long`	`LONG :: lval : clong;`
`byte`	`BYTE :: bval : cbyte;`
`char`	`CHAR :: cval : cchar;`
`float`	`FLOAT :: fval : cfloat;`
`double`	`DOUBLE :: dval : cdouble;`

ddl 6

```
productnode :: value : integer * integer;
```

Abstract data types The DDL of TTT offers 4 abstract data type constructors, the set type operator (`set`), the sequence type operator (`seq`), the map type operator (`->`) and the product type operator (`*`). These data type constructors have similar semantics as their counterparts within the VDM-SL specification notation. DDL offers a wide range of abstract data type operators, both destructive (e.g. set addition) and non destructive acting on sets, sequences, maps and products, including most of the operators defined in VDM-SL.

Union types The union type constructor `|` defines a type whose values belong to the set theoretic union of the given operand types. Union types form a convenient way to define classes of values that are further divided into distinguished subclasses.

ddl 7

```
form          = square | circle;
XYcoordinate = integer * integer;
square       :: size   : cint;
circle       :: radius : cint;
object       :: kind   : form,
                pos    : XYcoordinate;
```

For example, in specification 7 some simple definitions are given to model geometric forms on a canvas. A type is defined which may hold values that belong either to the type `square` or `circle`.

3 THE TTT C-LANGUAGE BINDING

The result of an application of TTT is a C header file called `tttlib.h` and a pre-compiled set of support functions, collected into an archive file, called `tttlib.a`. This section will explain the TTT generated C language data structure definitions and support functions. The support functions generated by TTT fall in three categories. First, for each record definition in the DDL specification a series of support functions are generated. Next a set of tree manipulating functions are generated and last, a set of support functions for manipulating ADT values is created. We will describe each category in some detail. For the sake of clarity we will address in this section C language types constructs as type classes, or short classes (e.g. int class, record class, union class etc.), and DDL constructions as types (e.g. ADT types, record types).

Record value functions For the support of DDL record definitions a wide range of C language functions are created, such as functions to create record values, initialization functions, access functions and test functions to establish their type. Each DDL record definition results in a C-language record class structure definition. Each record holds a series of standard attributes and a sequence of user defined fields. All C-language record structure definitions are bundled into one union class. To access the record values through this union, a generic pointer class is defined `tnp`, a *tree node pointer*, which is used throughout the TTT C-language binding. Values pointed to by the `tnp` values are either user defined record values or predefined record values like the ADT values or the DDL basic type values. Doublet values are tuple values containing both a data value and a DDL type indication of the data value. The origin of the doublet phenomenon can be traced back to compiler technology [7]. The interpretation of `tnp` values is only possible for record structure values, as each record value contains a type indication. As such record values form also a kind of doublet, by combining both a value (the sequence of fields) and a type indication. If a `tnp` value consists of a ADT value, the top node indicates the overall type of the value, being either a set, sequence, map or product value. The actual structure of the ADT value cannot be deduced from the top node. Whenever an abstract data type is included in a type expression for a record field , we lose the ability to predict the type of the field value. For these fields doublet values are created that hold a special indication of the type of the field value. This indication is used for dynamic type assertion, which will be explained in section 4.

Node creation and initialization Node creation is done by a call to the node creation function * `tnp mk_node (ttag XNodeTypeIdentifier)`. For each predefined and user defined record definition a type designator is defined by TTT as `XNodeTypeIdentifier`. A type designator has a C language type class

*The signature of all functions in this section will be presented in C language style.

`ttag`, which is a subclass of the C language integer class `int`. E.g. the make functions generated for the records resulting from specification 7 have the following C signatures.

```
tnp mk_object (tnp kind, tnp pos);
tnp mk_circle (int radius);
tnp mk_square (int size);
```

Values for the object record definition (7) created in terms like

```
mk_object (mk_square (3), mk_( mk_integer (1), mk_integer (2))),
mk_object (mk_circle (6), mk_( mk_integer (5), mk_integer (5)))
```

define tree values belonging to the record definition `object`. The values are defined through terms built from make functions. Their structure is depicted in figure 1. Here again we see the different usage of flat C type values and

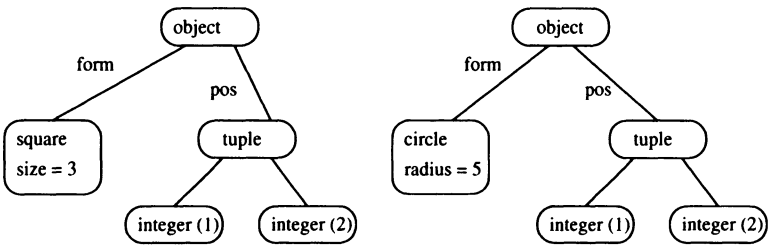

Figure 1 Two objects defined by terms

TTT basic type values. The `XYcoordinate` values are set by terms using the TTT integer type. The value attributes of the `square` and `circle` objects are set by terms using flat C language values. The union type operator is not in the structure of the object values. It is only part of the type definition and will be used to check whether the two object values have appropiate forms, during dynamic typechecking.

Access functions For each predefined and user defined field definition two access functions are created, one to set values to the field and one to get the field value. The syntax is `void s_`*`fieldname`* `(tnp node, fieldtype fieldvalue)`, the void function to set the value of field *fieldname* of tree node value `tnpvalue` to `fieldvalue`, and `fieldtype g_`*`fieldname`* `(tnp node)`, the value returning function to get the value of field *fieldname* from the tree node value `tnpvalue`. The set and get functions can be used to create tree node values instead of using terms consisting of make functions. Set and get functions, in combination with the standard `mk_node` function are more appropriate to generate

tree node values than terms built from specific `mk_` functions in the context of bottom-up tree value creation. For example, the term to create a value for the object type from specification 7

```
mk_object (mk_square (3), mk_( mk_integer (1), mk_integer (2)))
```

is equivalent to the following C-language fragment.

```
{
  tnp int1  = mk_integer (1);
  tnp int2  = mk_integer (2);
  tnp tuple = mk_node (XTUPLE);
  tnp sq    = mk_node (Xsquire);
  tnp obj   = mk_node (Xobject);

  s_left (tuple, int1);
  s_right (tuple, int2);
  s_size (sq, 3);
  s_kind (obj, sq);
  s_pos (obj, tuple);
}
```

As illustration a small part of a YACC specification is presented in figure 2 where tree node values are created in a bottom-up style.

Tree node type assessment For each predefined and user defined record definition TTT generates a boolean valued test function to assess its type. The syntax is `is_`*`NodeTypeIdentifier`* `(tnpvalue)` where *NodeTypeIdentifier* is the name of the record defined in the record definition and `tnpvalue` is the tree node value argument for which the type assessment is sought. First the assess function checks whether the tree node value is of the proper record kind, next it checks the type of the value by calling the type checking function `tassert`, which will be explained in section 4.

4 STATISTICS

We have performed several benchmark tests to investigate the timing characteristics of TTT. The analysis presented here is performed on the timing output and profiling data collected through the GNU C environment under the Linux 2.0 operating system. The benchmark test for which the data is presented had the following structure.

```
defs : def
    { tnp map = mk_node (XMAP);
      s_tdom (map, XTYPENAME);
      s_trng (map, XDEFINITION);
      mapadd (map, $1);
      $$ = map; }
    | defs SEMCOL def
    { mapadd ($1, $3);
      $$ = $1; } ;
def : isdef
    { $$ = $1; }
    | colcoldef
    { $$ = $1; } ;
isdef : typename IS typeexpr
    { tnp isdef = mk_node (XISDEF);
      tnp maplet = mk_node (XMAPLET);
      s_definition (isdef, $3);
      s_del (maplet, $1);
      s_rel (maplet, isdef);
      $$ = maplet; } ;
colcoldef : typename DEFINES fieldlist
    { tnp colcoldef = mk_node (XCOLCOLDEF);
      tnp maplet = mk_node (XMAPLET);
      s_fields (colcoldef, $3);
      s_del (maplet, $1);
      s_rel (maplet, colcoldef);
      $$ = maplet;
    } ;
```

Figure 2 YACC style bottom-up tree value creation

```
for i := 0 to 9 do
   set1 =  { e | 0 <= e < 2000.i }
   twrite set1; set2 := tread set1;
   tequiv (set1, set2); tdiff (set1, set2);
   tfree (set2); set2 := tcopy (set1);
   tassert (set1);
   tfree (set1); tfree (set2);
od
```

As such the overall benchmark test created 10 sets, ranging from 0 to 18000 elements. The call statistics of the benchmark (showing only the relevant call data) is as follows.

1	cachec2ttd	1	init_cache
1	init_t ime	10	gsize
10	tcopy	10	tdiff
10	tequiv	10	tsizeof
10	twrite	11	tread
20	tassert	30	tfree
80	unmark	90000	mk_integer
90002	setadd	180000	assert_GT
180010	get_lun	180010	write_node
180108	read_node	360020	g_lineno
360020	g_lun	360118	s_lineno
900369	mk_node	900369	n_init
900380	salloc	890002	g_val
250421	g_next	320240	s_mark
120944	g_tag	7020390	g_mark

The difference between the tree write and read operations is explained by the type tree cache initialization. The number of set related nodes totals 180010. Through the tree operations the sets are copied 5 times, resulting in 900000 node creations. The time characteristics of the overall benchmark test is presented in figure 3. The dimension of the time axis is in user seconds overall

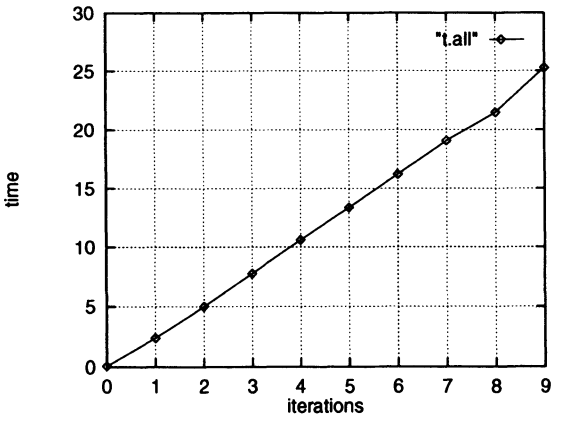

Figure 3 Overall timing benchmark

runtime. Figure 4 shows the difference in timing characteristics between the tree read and tree write operation. These operations are relatively more time consuming than the internal tree operations (see fig. 5) as file handling capabilities of the operating system are involved. The timing characteristics of the basic tree operations are presented in figure 5. As reference, the time needed to generate the sets (setops) is also presented. The time characteristics

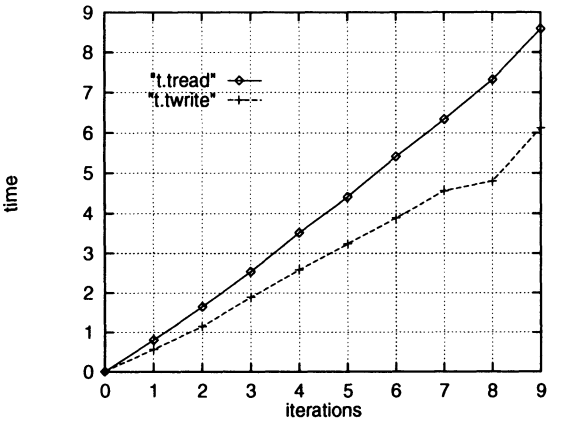

Figure 4 Tree read/write operations

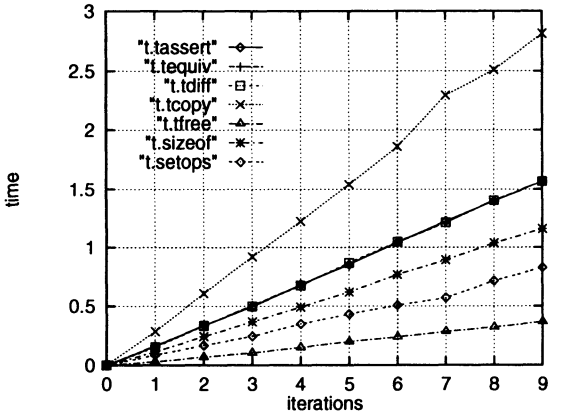

Figure 5 Tree operations

are produced on a 100 Mz Intel DX4 processor, running Linux 2.0. The same benchmark running on a 166 Mz Pentium Pro under Linux 2.0 produced a speedup of 33 %. Similar results were produced on a Sun Sparc-20 architecture running Sun Solaris 2.5.

We have run the same benchmark test on the AST tool. As AST has no capabilities for the set abstraction, a simple linear list based C application was added to the AST test version. The timing results of the total runtime in seconds are summerized in table 3. In both tools we see that tree reading and writing takes relatively the most execution time. Both tools apply a ASCII based format. The format used by AST results in enormous files. E.g. the file that stored the results of the last iteration took 75 MB external storage! The

Table 3 Comparison of AST and TTT facilities

tree facility	AST result	TTT result
reading	390.50	7.70
writing	385.05	5.55
type checking	148.53	0.91
freeing storage	0.095	0.320
test on equality	0.087	0.678
copying	0.079	1.313

figures shows that TTT's format is far more efficient. The external TTT file took 795 KB external storage. The in core manipulations are faster in AST.

5 CONCLUSION

TTT is born out of two different needs, (i) the need to have a ***simple*** tool for both abstract syntax manipulation and abstract data type value manipulation within the context of the educational program of the software engineering and compiler construction section of the faculty, and (ii) the need to have a fast and concise tool for abstract data type value manipulation, based on the VDM-SL abstractions for the tool constructing activities within the real-time laboratory of our section.

Our group is applying TTT within the context of a graduate course on compiler construction and in a large tool constructing project. The project is dedicated to the development of a structured operational semantics based analysis and prototyping environment generator. The generated environments are dedicated to specification and programming notations for real-time distributed processing systems and offer tools for analysis (model-checkers), visualization, simulation and interpretation.

The implementation of TTT abstract data type values is optimized for various forms of usage. We are incorporating different implementation models, ranging from very fast access models to very sleek representational models. Further we are incorporating a simple modularization scheme, that enables the usage of different sets of DDL definitions within a single application. Thus the DDL notation becomes modular. Currently we have no plans to implement a TTT data manipulation language (DML), such as a VDM-SL language subset. These subsets are already implemented e.g. like in the C++ binding of the IFAD VDM toolbox (based on [9]).

REFERENCES

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, Reading, MA, 1996.

[2] J. Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[3] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, pages 121–130, February 1992.

[4] J. Grosch. Ast - a generator for abstract syntax trees. Technical Report 15, Gesellschaft für Mathematik und Datenverarbeitung mbH, Universität Karsruhe, 1982.

[5] J. Grosch. Generators for high speed front-ends. *LNCS*, 371:81–92, 1988.

[6] C.B. Jones. *Systematic Software Development Using VDM, 2-nd edition.* PHI. Prentice Hall, 1990.

[7] J. van Katwijk and J. van Someren. Descriptors for Das. Technical Report 82-20, Faculty of Technical Mathematics and Informatics, Delft Univeristy of Technology, 1982.

[8] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1988.

[9] P.G. Larsen et al. The dynamic semantics of the bsi/vdm specification language. Technical report, IFAD, The institute of Applied Computer Science, Munkebjergsvaenget 17, DK-5230 Odense M, Denmark, August 1990.

[10] B Liskov and J. Guttag. *Abstraction and Specification in Program Development.* McGraw-Hill, New York, 1986.

[11] J.R. Nestor, J.M. Newcomber, P. Giannini, and D.L. Stone. *IDL: The Language and its Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[12] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets: An introduction to SETL.* Springer Verlag, Berlin, 1986.

[13] J.M. Spivey. *The Z Notation.* PHI. Prentice Hall, 1987.

[14] B. Stroustrup. *The C++ Programming Language.* 2nd Edition, Addison Wesley, Reading, MA, 1992.

6 BIOGRAPHY

dr.ir. Hans Toetenel graduated in 1984 at Delft University for the master of science program in technical mathematics and informatics. After a short period in industry, working for Philips Telecommunication Industry and AT&T as software engineer he re-entered the academic world, again at Delft university as a lecturer on programming languages. In 1992 he finished his doctoral thesis on formal specification of real-time systems. In 1993, he started working as associate professor on programming languages and compiler technology. His research interests are formal modelling, analysis and implementation of real-time and embedded systems.