ICFP 2009 experience-report submission

Experience Report: Putting an Oxymoron to Work

Using established functional-programming technology to implement TTCN-3

Michael Sperber
DeinProgramm
sperber@deinprogramm.de

Matthias Neubauer
intaris GmbH
Matthias.Neubauer@intaris-software.de

Abstract

The functional-programming community has produced mature tools for writing compilers: This paper describes our experience implementing TTCN-3, a programming language designed for writing test suites for programs and protocol implementations. Our TTCN-3 compiler is written in Scheme; it builds upon the Transformational Compiler that originated in the T and Scheme 48 projects, as well as the Essence functional LR parser generator, which is implemented using partial evaluation—both of which have existed for more than 20 and 10 years, respectively. Development was a straightforward experience—the only source of problems was the considerable complexity of the source language.

1. Introduction

In the universe of programming languages, the worlds of functional programming and TTCN-3 (ETSI ES 201 873-1 V3.4.1 2008-09) could hardly be further apart:

> The Testing and Test Control Notation Version 3 (TTCN-3) is an internationally standardized language for defining test specifications for a wide range of computer and telecommunication systems. (Willcock et al. 2005)

As such, TTCN-3 is a concurrent, but otherwise traditional procedural language, with various special features to make it suitable for writing test suites, including an elaborate type system with subtyping, and an extensive framework for doing pattern matching on values. The TTCN-3 standard specifies an entire infrastructure, including graphical and tabular notations for TTCN-3 programs, and significant parts of the run-time system. The complexity of the language and its run-time system make implementing TTCN-3 a daunting task, especially as the language specification does not employ the idioms and language commonly used by the programming-language community.

This experience report describes our efforts implementing a TTCN-3-to-C compiler in Scheme at intaris, a Freiburg-based software company. As the goal was to produce a functioning compiler with minimum effort, we chose to re-use existing compiler infrastructure as much as possible. The crucial re-used bit is the the Transformational Compiler, which was originally part of the T project but has survived as the core of the Pre-Scheme compiler used to compile the VM of Scheme 48. Moreover, we used the Essence parser generator, along with a newly developed scanner generator to develop the front end.

[Copyright notice will appear here once 'preprint' option is removed.]

Once we had chosen these base technologies, development was essentially a straightforward exercise: The Transformational Compiler with its time-proven CPS representation forms the backbone of the system, and greatly helped structure the compiler itself. The CPS representation also lays the groundwork for the implementation of concurrency, as the C run-time system employs user-level threads and needs to pre-empt and re-enter the running program.

Consequently, there were hardly any notable surprises to report. The main issues that arose during the development of the compiler had to do with the (from a programming-language viewpoint) idiosyncratic nature of TTCN-3. Writing the compiler itself took about 80 man-days, including the scanner generator—given that the relevant parts of the standards number over 400 pages, and compiler development is typically priced at thousands of dollars per page of programming-language standard, this was very little effort. In addition to the advantage in expressiveness, Scheme 48 supports fast development through quick turnaround: The system lets the programmer load, run, and change the TTCN-3 compiler in place in a controlled manner, which gives further boost to productivity.

Our work demonstrates that even in the cutting-edge field of functional programming, shopworn but rock-solid technologies exist that are worth considering for new projects.

2. TTCN-3

TTCN-3 has been developed by the Methods for Testing and Specification Technical Committee (TC-MTS) at the European Telecommunications Standards Institute (ETSI). The predecessor of TTCN-3, TTCN-2 ("Tree and Tabular Combined Notation"), was a limited tabular notation for expressing tests, and incorporated protocol-specific terminoloy. In constrast, TTCN-3 is a full-fledged programming language, targeted at specifying and writing test suites for system components, protocols or systems in distributed settings.

The underlying idea of TTCN-3 is that test suites for a specific "system under test" (or *SUT* for short) or protocol are specified and written in TTCN-3, independently from the implementation platform of the SUT, the programming language used to implement the SUT, the internals of the TTCN-3 run-time system, or other specifics resulting from deploying the system to a specific setup. Public TTCN-3 test suites exist for IPv6, DHCPv6, and SIP.

2.1 The TTCN-3 Programming Language

TTCN-3 is an imperative programming language with concurrent-programming features based on the actor model (Hewitt et al. 1973), and additional constructs for writing test cases. Its syntax is loosely based on other popular imperative programming languages that distinguish between expressions and statements; it draws on elements from the Algol family (such as Algol-style assignments) and C (such as curly braces as block delimiters).

TTCN-3 has a partially static type system with a set of base types including integers, floats, and character strings, with a special base type `verdict` for denoting test results, and with user-defined composite types. Composite types include ordered and un-

```
module MyTestModule {
  type record MyMessageType {
     integer    seqNum,
     record of charstring data
  }
  type port MyPort message {
     in MyMessageType
  }
  type component MyTestComponent {
    timer       timer1;
    port MyPort port1
  }
  template MyMessageType msgTemplate := {
    seqNum := (100 .. 200),
    data   := {*, permutation("hans", "fritz", "franz"), ?}
  }
  testcase myFirstTestcase() runs on MyTestComponent {
    timer1.start(60.0);
    alt {
       [] port1.receive(msgTemplate) {
         setverdict(pass);
       }
       [] timer1.timeout {
         setverdict(fail);
       }
    }
  }
}
```

Figure 1. Example TTCN-3 test module

ordered product types (called *records* and *sets*, respectively), sum types (called *unions*), and a plethora of ordered and unordered aggregate types. The language also has a notion of subtyping atop the static types, only part of which is checked statically—the rest is done at run time. The programer defines a new subtype of an already existing type by restricting its values, e.g. via subranges, size restrictions, or patterns similar to regular expressions.

The TTCN-3 programming language also incorporates a run-time pattern-matching mechanism. Patterns—called *templates*—are run-time values. Templates specify the shape of a value, with restrictions on the variable parts. (In contrast to ML-style pattern matching, a TTCN-3 template is not a binding construct.)

TTCN-3 has three different parameter-passing mechanisms, corresponding to call-by-copy, call-by-value-result, and call-by-reference. In contrast to many other languages supporting a call-by-value evaluation strategy, TTCN-3 has no explicit references—even structured data types such as arrays are copied on assignment. The absence of references incurs significant run-time overhead, but also obviates the need for automatic memory management.

Actors are called *components* in TTCN-3. The SUT is represented by a virtual *system component*; it communicates via messages with the test suite. Components declare communication endpoints called *ports* that support asynchronous sending and receiving. The language has a selective-communication mechanism called `alt` for formulating complex communication patterns. The `alt` construct addresses some of the same problems as the selective-communication combinators of Concurrent ML (Reppy 1999), but is less compositional and entirely based on polling. An `alt` construct takes a snapshot of the various synchronization objects (ports, timers, components), and then refers to that snapshot to ensure consistency among alternatives. A semi-formal "operational semantics" in the TTCN-3 standard uses annotated flow charts to specify the concurrency substrate.

TTCN-3 has a simple module system: Modules group named definitions, which can be imported by other modules.

Figure 1 contains the code of an TTCN-3 module. It shows one module `MyTestModule`, which contains five definitions:

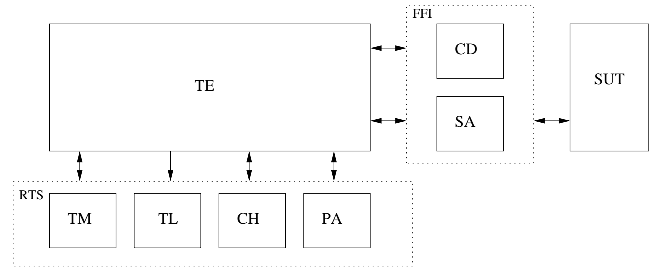

Figure 2. TTCN-3 run-time environment

`MyMessageType` is a user-defined record type that contains an integer and a sequence ("record of") of strings.

`MyPort` specifies the type of a communication endpoint, which can receive incoming values of type `MyMessageType`. Components of type `MyTestComponent` own a communication port `port1` and a timer named `timer1`. A timer is started with a specific duration and eventually signals the end of that duration.

The template `msgTemplate` matches values of `MyMessageType` type whose `seqNum` field is an integer value in the range from 100 to 200, and its `data` field is a sequence that begins with an arbitrary number of arbitrary strings (`*`), continues with a permutation of three specific strings, and ends with an arbitrary string (`?`).

The last definition defines a test case named `myFirstTestcase`, which runs on components of type `MyTestComponentType`. The first statement of the body of the testcase starts the timer initialized with a maximal duration of 60 seconds. The `alt` statement loops until either a message matching template `msgTemplate` is received over `port1`, or the timer `timer1` expires. The `setverdict` statement sets the test outcome accordingly.

2.2 TTCN-3 Run-Time Environment

The TTCN-3 standard defines different kinds of APIs that constitute a test environment for a SUT. From a programming language perspective, they can be broadly categorized into two kinds of functionality:

- **parts of the language runtime system (RTS)** startup primitives (TM), logging primitives (TL), primitves for actor communication (CH), and timer primitives (PA)
- **FFI bindings for the SUT** data serialization/deserialization code (CD), primitives for actor communication with the SUT (SA)

A TTCN-3 compiler assumes the existence of implementations of these APIs. The standards defines APIs for C and Java bindings.

Figure 2 gives an overview of the different components involved in a running TTCN-3 test system. The TE ("test executable") component is compiled TTCN-3 program code. The CH API is devised such that it is possible to deploy the TE—or parts of it—on distinct hosts with message communication. TTCN-3 code never directly communicates with the SUT. All communication is routed through the SA, applying the SUT-specific serialization/deserialization (CD) in between.

3. Corporate Environment

intaris GmbH is a small software development and consulting company in Freiburg im Breisgau, Germany. A major focus of the company is in software testing, mainly for the embedded market. Typical customers are companies from the industrial sector, in areas such as process automation or railroad engineering.

TTCN-3 is well-established in the testing business, especially for protocol testing in communication technologies. The needs

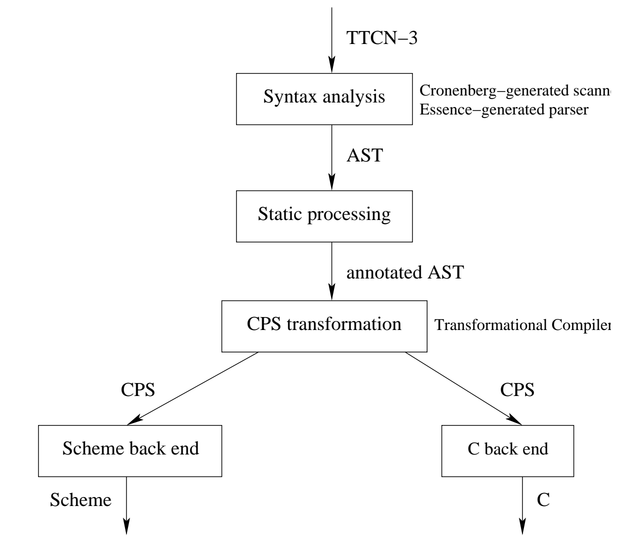

Figure 3. Compiler architecture

of typical customers of intaris GmbH with respect to a TTCN-3 test environment gave rise to the development of a custom-made TTCN-3 compiler with its own requirements, which include:

- The target language of the compiler has to be (portable) C, because C/C++ is the only available option for many embedded devices (besides assembler).
- Some target platforms run without an operation system. It must be possible to generate code that does not require an existing task/thread scheduler.
- It is essential for some target platforms to make predictions about the resource usage of a test environment to be deployed.
- Solid software development methods should be used so that software building on the compiler can meet substantial safety and integrity standards.

None of the available TTCN-3 implementations meets all of these requirements. Moreover, no open-source implementation exists that could be modified to meet them.

4. Compiler Implementation

Figure 3 shows the overall layout of the compiler, which is completely traditional; the stages are described in the following subsections. The Scheme backend was not part of the requirements, but greatly simplifies testing, as it is possible to generate code, compile and run it all within the running Scheme 48 session without having to produce intermediate files, invoke the compiler and run the resulting executable.

4.1 Syntax Analysis

The syntax of TTCN-3 is quite complex: The grammar listed in the TTCN-3 specification (ETSI ES 201 873-1 V3.4.1 2008-09) contains more than 1000 productions. It is highly ambiguous: The grammar assumes that an identifier binding has already been resolved prior to parsing to some degree. Here is an excerpt:

```
161. FunctionIdentifier ::= Identifier
177. FunctionRef ::= [GlobalModuleId Dot]
       (FunctionIdentifier | ExtFunctionIdentifier)
270. ExtFunctionIdentifier ::= Identifier
```

Thus, in its raw form, the grammar is unsuitable for consumption by a parser generator, and the main effort of constructing a parser was spent on transforming the grammar into SLR(1) form.

This process was aided by the parser generator used: Essence is an LR parser generator, which is itself automatically generated from a generic LR parser via partial evaluation (Sperber and Thiemann 2000). Essence was originally built as a research prototype to validate the effectiveness of partial evaluation, but proved well-suited to the task at hand: The interpretive Essence parser can run directly, accepting the grammar and the input, and it starts parsing immediately, without constructing parser or lookahead tables first. This reduced turnaround time compared to traditional generator-based or macro-based approaches (Flatt et al. 2004)—and many turnarounds were needed to massage the grammar into SLR(1). The (purely functional) code of the generated parser is extremely fast.

We developed the scanner with conventional techniques: We first wrote a minimalistic scanner generator (called *Cronenberg*) to go with Essence, and used it to build the TTCN-3 scanner.

4.2 Static Processing

The abstract syntax tree is represented using a generic data type called *node*; the implementation of nodes borrows heavily from a similar data structure used by the Scheme 48 byte-code compiler. Each node carries an *operator*, a source location, an arbitrary number of arguments (each of which may be another node, or some constant), and a property list mapping names to arbitrary values. Dealing with nodes, Scheme's macros and latent typing come into full play: Various macros define automatic-deconstruction facilities and pattern patching. Latent typing enables attaching arbitrary information to a node as static processing progresses. Moreover, it allows various types of generic processing, such as free-variable analysis, syntax-tree serialization, or automatic conversion to a dot input file to draw a syntax tree.[1]

The parser produces an abstract-syntax tree represented as a node. Nodes operator types are defined like this:

```
(define-node-type send-statement (exp param to-clause))
```

Procedures can then use a variety of constructs to dispatch on node operators, and deconstruct the node. Here is a corresponding example that uses the `node-type-case` macro:

```
(define (process-statement global-env local-env statement)
  (node-type-case statement
    ((send-statement port param to-clause)
     (let ((port-type (process-port/type
                       global-env local-env port)))
       (process-expression/port-argument
         global-env local-env port-type 'out param)
       (process-to-clause global-env local-env to-clause)))
    ...))
```

Static processing comprises a total of 8 passes over the source code. These are needed because the top-level constructs of the language are interdependent in various ways, and do not need to appear in order in a TTCN-3 program. In particular, types may include restrictions (range restrictions, value lists, regexps etc.) described by static expressions, which in turn are typed ... This requires careful staging and various topological sorts along the way. Thankfully, expression processing needs only one pass to perform type checking and compute statically available values. The types and values are attached to the abstract-syntax nodes as properties.

[1] Statically-typed languages offer essentially two choices to replicate this kind of representation: 1. Each operator corresponds to a summand in an algebraic data type, which allows static type checking of pattern matching, but makes generic processing difficult. 2. A generic representation abandons static type checking, and requires sum types for every data type needed by the compiler—this clutters the code with artificial pattern matching.

4.3 CPS Transformation

The central part of the compiler is the Transformational Compiler (TC) (Kelsey and Hudak 1989) of the T project (Kranz et al. 2004). The TC uses a single intermediate representation for several different stages of the compiler. In Scheme 48 itself, the TC is both the central part of the Pre-Scheme compiler (Kelsey and Rees 1995), which compiles the VM to C, as well as the native-code compiler. Still, the TC is in no way tied to Scheme as the language to be compiled. (The original TC was used to compile Pascal.)

In its latest incarnation as part of Scheme 48, the TC uses a CPS intermediate representation. In contrast to CPS representations that use different top-level constructs for regular abstractions and continuations, the TC represents all λ nodes by a single data type. Tags attached to to the λ nodes divide them into three classes, depending on how the value of the form is used. Continuations are λ forms passed as continuation arguments to primops. Jump targets are λ forms whose calling points have been identified and which are all within the same procedure λ. All other λ forms are procedures.

The CPS nodes form a graph which is manipulated imperatively, avoiding the need to reconstruct an entire tree when a transformation affects only a small parts of it. This approach, as well as the uniformity of the representation makes various transformations such as contification particularly easy to implement. (The TC representation has since been re-discovered and shown to be superior to direct-style representations in several ways (Kennedy 2007).)

The TC provides various tools for dealing with the CPS representations, including a library for constructing CPS transformations, a pretty printer, and a generic optimizer which can be parameterized with simplifiers for primops of the particular language being compiled. These proved valuable for developing the CPS transformation at the heart of the TTCN-3 compiler. Generally, using CPS helped resolve the semantics of some complicated language constructs, specifically that of `alt` and snapshots, as CPS makes explicit the implicit changes in the control flow of these constucts.

Still, the complexity of the language makes for a rather large CPS transformation (4000loc), despite the prior expansion of the code to core constructs. The complex semantics of references, mutation, and argument passing complicate the task enormously.

4.4 Code Generation

The code generated by the Scheme backend is just the elaborated CPS code; the code generator is less than 200 lines. It is so simple because the target language is higher-order, has automatic memory management, and supports threads natively.

The C code generator is more interesting, as C does not support closures directly, and the code has to meet external requirements: Memory management must be explicit, and the code must run under a portable, user-level pre-emptive thread system. The first requirement is met by a liveness analysis that determines when locations die, and assigns frame slots to them. The code generator uses the results of the analysis to insert explicit deallocation into the code. The second requirement means that the output code cannot use the C stack for function calls, but must manage continuations explicitly. To that end, the code uses the classic trampoline technique (Tarditi et al. 1990). Again, CPS helps structure the process, as the explicit continuations correspond to the entry points of a function in the generated code.

5. Run-Time System

The run-time system developed for the Scheme backend proved enormously valuable in developing its counterpart in C. As the Scheme code is a direct transliteration of the CPS output code, the Scheme run-time system neatly separates code-generation issues from the run-time infrastructure, whereas, with the C code, both are more closely intertwined. This helped with experimentally clarifying the TTCN-3 execution model. It was particularly valuable in implementing the complicated template matching mechanism: We initially developed the code for handling sequence patterns—which includes constructs such as the Kleene $*$ with length restrictions and arbitrary permutations—in Scheme, performed lambda lifting (Johnsson 1985) on it and transliterated the result to C.

6. Conclusion

When the project originally started, it did not seem such a terribly large task, and we probably would have started on it even if requirements had forced us to use a more conventional language. Along the way, the complexity of TTCN-3 gradually revealed itself, and it became clear that conventional technologies would have stretched the required effort far beyond the resources available to us. The task only became tractable through the use of functional programming, the interactive Scheme 48 development system, and available infrastructure for constructing compilers: Fortunately, Scheme, Essence and the Transformational Compiler had not accreted any digital mould, and were up to the task.

References

ETSI ES 201 873-1 V3.4.1 (2008-09). *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, 2008. `http://www.ttcn-3.org/StandardSuite.htm`.

Matthew Flatt, Benjamin McMullan, Scott Owens, and Olin Shivers. Lexer and parser generators in Scheme. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the Fifth Workshop on Scheme and Functional Programming*, pages 41–52, Snowbird, October 2004. Indiana University Technical Report TR600.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, Standford, CA, August 1973. William Kaufmann. ISBN 0-934613-58-3.

Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Functional Programming Languages and Computer Architecture 1985*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, January 1989. ACM Press.

Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

Andrew Kennedy. Compiling with continuations, continued. In Norman Ramsey, editor, *Proceedings International Conference on Functional Programming 2007*, pages 177–190, Freiburg, Germany, October 2007. ACM Press, New York. ISBN 978-1-59593-815-2.

D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In Kathryn S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection*, pages 175–191. ACM, April 2004. SIGPLAN Notices 39(4).

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.

David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. Wiley, 2005.