Assaulting the Tower of Babel: Experiences with a Translator Writing System

by

Harvey Abramson
William F. Appelbe
Mark Scott Johnson

Technical Report 77-12

1977 November

Department of Computer Science
The University of British Columbia
Vancouver, British Columbia V6T 1W5

Abstract

TRUST is a translator writing system (TWS) which evolved from several available TWS components, including an LR(k) parser generator and a lexical scanner generator. The design and historical development of TRUST are briefly presented, but the paper is primarily concerned with relating critically the experiences gained in applying the TWS to various practical software projects and to the classroom environment. These experiences lead to a discussion of how a modular TWS should be designed and implemented.

KEY WORDS: Compiler construction Compilers Generators
Translator writing systems TRUST

This work was supported by the National Research Council of Canada through operating grants A-9132 and A-7447.

INTRODUCTION

TRUST is the name of the translator writing utility system developed in the Department of Computer Science at the University of British Columbia. It was not the original intent to develop a translator writing system. Rather, as a result of graduate student research, several components of a TWS (e.g., an LR(k) parser generator and a lexical scanner generator) were available. Since these projects represented considerable investments in time and money, it was agreed they ought not to expire when the initial projects terminated. An attempt was made to build the remaining parts of a TWS (e.g., components to construct and traverse trees and perform semantic actions) and to design some method of combining and using these components. Hence TRUST, like Topsy, "just growed".

This paper describes the historical development of TRUST to its current state, enumerates uses to which it has been put, evaluates the degree of success of these applications, criticizes the entire effort, and draws conclusions concerning the feasibility of constructing such large-scale systems in a university environment.

OVERVIEW

The oldest components of TRUST are an LR(k) parser generator and a lexical scanner generator based on regular expressions. The parser generator was constructed by David Ramer as part of an ALGOL 68 implementation.[7] It was needed to generate an LALR(3) parser for an ALGOL 68 translator, but was applicable to other projects. The scanner generator was constructed as part of the thesis research of Ted Venema, who was interested in the general problem of the modular design of translators.[11][12]

Concurrently, there was interest in the department in syntax directed translation and the specification of programming language semantics by interpreting sets of strings attached to the nodes of derivation trees. Previous work on syntax directed translation[1] and macroprocessing[2] led to the design and implementation of TOSI, a tree oriented string interpreter.[3][9] TOSI was initially used by Tom Rushworth to generate code for an ideal ALGOL 68 machine as part of his thesis research.[10]

These main TWS components, a parser generator, a lexical scanner generator, and a mechanism for code generation, were thus separately available. It was possible, though just barely, for a skilled hacker to use them to construct translators. It was decided then to clean up the available components, create other necessary ones, and impose a communications structure between them so translators could be written more easily. The system so constructed is described in the _TRUST User's Guide_.[13] Currently, TRUST is in a state of dormancy. While there are some users, the system is not being actively developed

and cnly a minimum level of support is being maintained.

Communication mechanism

A TRUST-generated translator is a democracy of modules. The TWS contains components to generate lexical scanners, lookup routines, parsers, tree builders for intermediate representations, and semantic routines. The resulting translator is a democracy in that no module may control another, nor may any module directly request anything of another. All requests for actions by another module, and all replies to such requests, are channeled through an _intermodule communicator_ (IMC). Should any operating system dependent requests be made, these too are channeled through the IMC to a special _module system communicator_ (MSC). The basic structure of a TRUST-generated translator is shown in Figure 1.

Any number of special-purpose modules may be generated by TRUST and interconnected. There may, for example, be an LR(k) parser, a simple precedence parser, or an LL(k) parser. The user selects the desired modules and links them together through a user-written IMC. If at any time a module must be replaced, a change to the IMC should suffice to effect the replacement.

Since modules cannot communicate with each other directly, the user must establish a communication network in the IMC. There is attached to each module a _communication block_ which is passed between the IMC and the modules. Associated with each TRUST-generated module is a unique COMMBLOCK through which all communication with other modules must be accomplished. The block is divided into three sections: an event number, a parameter list, and ninety words of local storage.

The _event number_ of the COMMBLOCK determines what acticn is to be performed by either the module (when control is transferred from the IMC to the module) or the IMC (when control is transferred from the module to the IMC). The writer of the IMC must know the event numbers associated with each module. The _parameters_ of the COMMBLOCK communicate information between the module and the IMC. A lexical scanner, for example, may require the address and length of an input buffer, and may provide the address, length, and type of a lexical token. The _local storage_ of the COMMBLOCK is reserved for the associated module and may be used as required (e.g., for storing local variables). The local storage must not be accessed by any other module.

EXPERIENCES

Each of the module generators currently supported by TRUST has its own specification language. For example, there are languages to specify IMCs, lexical scanners, and LR(k) grammars. TRUST itself was used to generate recognizers for each of these admittedly simple languages. Nevertheless, this does demonstrate the possibility of applying a TWS to itself to bootstrap to more complex languages.

The first major attempt to apply TRUST to producing a useful translator was the development of a cross-assembler for the HP2100MX. This presented a practical challenge since the assembly language was quite unlike the high-level structured languages for which TRUST was principally designed. Initially, TRUST was used to develop a syntax checker for the assembly language, then this was extended to produce relocatable binary object code for the HP2100MX. All the modules except for the semantics were generated using TRUST. The semantic module was written in PL360 and linked directly to the IMC.

The greatest difficulty in implementing the syntax checker was encountered in defining the scanner and parser. The format of the assembly language was strongly card-oriented and not directly suited to a scanner based on regular expressions. Rather than writing a scanner by hand, the format of assembly instructions was restricted to make it compatible with the generated scanner. Thus, all comments were required to be prefixed by asterisks, and Hollerith constant lengths were ignored. Major problems were also encountered when the syntax checker was extended to a full cross-assembler. The format of the relocatable binary object code was clearly unsuited to single pass translation, and it was difficult to decide on an appropriate structure for the semantic phase. At this point the project ground to a halt because users were dissatisfied with the language incompatibilities and restrictions, though the implementors judged the project a success since it demonstrated the feasibility of using TRUST for an unstructured language. Eventually, a full cross-assembler was written directly in BCPL, with less effort than would have been required to generate a full cross-assembler using TRUST.

Components of TRUST were used to generate a bootstrap version of a C translator.[8] The components used were the scanner, parser, and lookup routine generators. The user found the TRUST tree builder to be unsatisfactory since it generated far too many trivial nodes (i.e., nodes with one input and one output arc) and the tree header nodes generated were unnecessarily large. He therefore replaced the TRUST tree builder with one of his own. Similarly, he found the TCSI-based semantics unsuitable and wrote his own semantic routines. He was able, using the generated TRUST modules together with his own, to write a bootstrap version of a C translator. This, of course, was for a subset of C, but a subset substantial enough to write

a full-scale C translator. This latter translator is now essentially operational.

The language B0 is the first of a sequence of proposals culminating in a language suitable for teaching programming to beginners.[5] The language is fairly simple, and an attempt was made to generate a translator for it. The attempt was severely constrained by the pressure of other work and by limited computing funds. Using TRUST, a translator from B0 to ALGOL 68 was designed and partially completed, at least enough to show that there were no insurmountable difficulties preventing completion, given sufficient time and money.

The principal obstacle to the completion of the B0 translator was the implementation of TOSI. In theory, a set of strings to control code generation should be read at translation time so that minor changes can be made to their definitions during the course of semantic development. Despite this, the TOSI implementors decided to compile the semantic strings into the generated semantic module to avoid the substantial overhead associated in performing transput with the local operating system when the generated translator is executed. This decision proved disastrous since any slight change to the set of semantic strings required the complete regeneration of the semantic module. Thus, although this made the actual running of the generated translator less costly, the cost of developing the semantics was high due to frequent expensive module regenerations. For this reason, the B0 project was dropped after a few weeks of intermittent effort.

Components of TRUST were incorporated into thesis research in the development of an interactive high-level debugging system.[6] This involved two aspects: the use of TOSI as a stand-alone interactive macroprocessor, and the construction of a syntax checker. The debugging system, called RAIDE, is controlled by the commands of a debugging language, called DiSpeL. Since DiSpeL looks very much like a macroprocessing language, it was initially implemented by extending TOSI to accept strings interactively and by defining the primitive RAIDE actions as TOSI functions. This use of a TRUST component demonstrated the ability to incorporate an interpretive module into a translator even though TRUST was designed primarily to accommodate generated modules.

The second application of TRUST to RAIDE was the development of a syntax checker for DiSpeL programs. This required the generation of scanner, lookup, and parser modules. The result of executing these modules was a parse trace of some input DiSpeL program; no semantics were associated with the program. As a byproduct, a general-purpose IMC was written to control syntax checkers for other languages. By combining the syntax checker with the interactive TOSI component, it should be possible to generate a translator for DiSpeL, though this was not done due to time constraints.

The most ambitious project relying on TRUST was the implementation of GRAIL, a semantic representation for the translation of high-level algorithmic languages.[4] GRAIL is a graphical program representation designed to simplify the development of translators. Instead of writing a translator directly from a high-level language into object code, the language implementor defines the translation of the source language into GRAIL and the subsequent translation of GRAIL into object code. The definition of the translation of the source language into GRAIL is compiled to produce input files for the TRUST module generators. Thus, the implementation of GRAIL constituted a semantic extension to TRUST.

The primary difficulties in using TRUST to implement GRAIL arose because of the incomplete documentation and poor debugging facilities of TRUST, especially in TOSI. Undocumented constraints were discovered in both the number of productions in the source language BNF and the total size of TOSI strings. These limited the implementation to developing translators for subsets of high-level languages. The TOSI semantics were found to be extremely expensive and tedious to debug since a minor syntax error invariably caused the semantic generator to fail without any indication of the source of the error. Debugging facilities had to be built into the IMC, written in ALGOL-W, which grew in size and complexity as more of the semantics were moved into it from TOSI. Despite these difficulties, the implementation was successful in demonstrating the use of GRAIL as an extension to TRUST.

Pedagogical experiences

An attempt was made to use TRUST in an undergraduate course on compiler construction. This proved to be premature since, although it had been shown useful for several graduate research projects, TRUST was not sufficiently documented and debugged for use at the undergraduate level.

For example, the lexical scanner generator adapted for TRUST was never completely implemented. Documentation of the actually implemented generator neglects to mention some of these restrictions. Thus, students followed the manual's specifications and rediscovered known bugs and misfeatures. When these deficiencies were made public, however, the students managed to incorporate TRUST-generated scanners into translators for Algol-like languages which they designed themselves. If any TWS is to be useful at the undergraduate level, it should be designed with a high degree of system security, error diagnostics, and complete and accurate documentation.

Criticisms of TRUST

Since the majority of users of a TWS such as TRUST are language implementors with broad backgrounds, they are almost certain to be highly critical of its design and development. In analyzing these criticisms, it is important to avoid basing them on minor aspects of the TWS, such as the format of the commands for generating modules. It is also equally important to avoid criticizing the TWS for applications which are outside its design goals. Consequently, this section concentrates on the design features which caused the most significant problems.

Perhaps the greatest difficulty comes in linking together the modules using the IMC generator. Although it does permit the language implementor to choose between coroutine and procedure modes of control, the IMC generator does not permit a controller to do anything other than transfer control to another module. The only facility for conditional transfer of control is the selection of a segment of code based on the event number, which must be performed when a module transfers control back to the IMC. Similarly, there is no facility for assigning expressions to parameters within communication blocks. In the IMC, parameters to a module must either be integer constants or parameters copied from another communication block. In principle, all these limitations can be overcome by writing an external module and an associated controller to perform the desired operation, but this becomes tremendously laborious and tends to convert the IMC into a nightmarish Gordian knot. In practice, nearly all users chose to write their own IMCs in high-level languages such as PASCAL or ALGOL-W. In spite of this, it remained necessary to imitate the structure of the IMC expected by all TRUST-generated modules. Even with only a few modules communicating via the event mechanism, it is extremely difficult to trace the flow of control between modules or to predict the effects of a change. The majority of users simply took a working IMC and modified it for their own use.

Another flaw in the design of TRUST is the interdependence of modules. In principle, a modular TWS should permit each module to be written, debugged, and tested separately. Nevertheless, even a minor change in one TRUST-generated module can force other modules to be rewritten. For example, the parser generator is closely linked to both the scanner and semantic generators. If the order of parser rules is altered, the semantic routines must be retranslated with new procedure numbers assigned. Similarly, if a new terminal is introduced into the parser productions, the scanner must be rewritten to assign a new sequence of lexeme numbers since the parser itself determines the ordering of terminal symbols alphabetically. This led to the ludicrous situation of users introducing new terminals such as ";", with the label "zemicolon" rather than "semicolon", to simplify rewriting and retesting the scanner. Nevertheless, at least part of this problem is due to the lexical scanner having been adapted for TRUST rather than having

been designed as part of TRUST.

IMPROVEMENTS

Perhaps the most critical need in developing a large software project such as a TWS is to integrate the system design. It is extremely frustrating for the language implementor to have to learn a new protocol for each module and then to have to juggle them together to build a system. Ideally, a modular TWS should force the implementor to use a top-down structured programming approach, rather than simply interfacing modules to produce a translator. Since TRUST was designed from the bottom up, it suffers from some severe design defects because of incompatibilities between modules. For example, it should be possible to redefine the productions of the parser generator without having to rewrite and regenerate the scanner and the lookup modules. Ideally, the IMC should be defined before any modules are generated, with stubs being substituted for missing modules. The current IMC structure simply provides a protocol for interconnecting existing modules. Communication between modules should be based on a user-oriented protocol, rather than on magic numbers passed from one module to another in series. The system should also enable each module to be written, generated, and tested separately once the overall structure of the translator has been defined. The IMC should provide the user with data and control structures which facilitate the construction of a translator to suit the user's own application. Although it is appealing to define a very simple universal model for the IMC, each user invariably prefers a different structure. Implementing these in terms of a simple communication mechanism such as TRUST provides is analogous to programming a Turing machine.

In TRUST, operating system dependencies are well isolated by means of the MSC. Nevertheless, the design of the MSC makes system requests somewhat cumbersome. A well-structured TWS should allow the implementor to define a hierarchy of system procedures which can be replaced and called directly by each module.

Although a TWS is highly dependent on its operating environment, emphasis should be placed on a reasonably portable implementation. Because TRUST is a gargantuan ensemble of PL360 programs, many of which rely on local operating system routines, it is difficult to move TRUST to another operating system, and inconceivable to adapt it to another machine. Consequently, the development of TRUST has halted since few people in a university environment are prepared to work voluntarily on software which they are unable to transport.

The TWS should be written in a portable, high-level algorithmic language such as BCPL or PASCAL, with the system dependencies as well-defined and isolated as possible. The TWS source language should be compatible with the range of TWS input

languages so it can be used to generate a translator for the TWS source language. In this way the TWS can be implemented on a new machine by bootstrapping. In this process, a translator is generated for the TWS source language and the new machine using the existing TWS, then this translator is used to compile a version of the TWS which will execute on the new machine.

Ideally, a TWS should provide the user with a wide range of aids for debugging and testing translators. At present, TRUST provides only rudimentary debugging aids. Each individual module may have its own facilities for tracing and debugging, but there is no overall debugging facility for the IMC. The TWS should provide the user with both meaningful trace and snapshot facilities to enable the sequence of module calls to be readily monitored. This can be done by providing a facility for generating an interactive version of the IMC, together with a set of dummy modules. An interactive IMC permits the user to generate a test translator which can be monitored and modified interactively.

CONCLUSIONS

In spite of the above criticisms, TRUST has proved to be a useful (though cumbersome) research tool. The development of TRUST, moreover, has led to the following conclusions.

The development of large-scale systems by patching together existing and newly written components under a rigid monitor is a poor methodology. Ideally, large systems should be designed and implemented top-down in the approved structured fashion, and system documentation should be written in the process. Universities are the primary proponents of the structured approach to the construction of programs, but ironically have been generally unable to apply the approach to large-scale research projects.

The structured approach to the construction of large systems requires not only a discipline of programming, but also an external discipline of programmers. Some restraint is required to insure that someone writes each specified part of some system. In a university environment, nonetheless, people generally choose to work on projects only out of intellectual or academic interest. It is difficult to assemble at one time a large team of researchers (say six or more) who will maintain interest in the tedious portions of important projects. Students complete their degrees and leave for jobs; faculty may be distracted by smaller scale, completable projects of their own. Moreover, when a large project is finally completed, the burden of system maintenance is likely to fall upon the users since the implementors are pursuing other interests. It is tempting to bemoan such a situation and compare it to industry where a more authoritarian discipline at least gives the possibility (though not necessarily the actuality) of applying the structured approach to large-scale systems. Industry, however, is rarely interested in research projects which lean

heavily on theoretical aspects of computer science; and industry too has substantial investments in existing software which prevent initiation of new projects.

University computer science departments have been slow to enter certain important research areas, such as the construction of practical translator writing systems or the design and implementation of large-scale database and information retrieval systems. The reason for this is neither lack of interest nor lack of funds (though the latter may play a part), but the nature of the university itself. Since the organization of universities is unlikely to change,* it appears that large-scale research projects started from scratch will continue to be avoided, and that most large projects in universities will have a patchwork effect. Alternatively, the question may be asked if there are not effective methods of organization other than the hierarchical. The question of the organization of large ongoing projects in a university environment requires further study.

ACKNOWLEDGEMENTS

The authors are indebted to the designers and implementors of TRUST, especially Ted Venema and Tom Rushworth, and to the various TRUST users, including Don Thomson, John Peck, and Peter van den Bosch, whose experiences shaped much of the content of this paper.

REFERENCES

1. H.D. Abramson, _Theory and Application of a Bottom-Up Syntax-Directed Translator_, Academic Press, New York, 1973.

2. H.D. Abramson, 'A syntax directed macro processor', _BIT_, vol. 14, 261-272 (1974).

3. H.D. Abramson, T.B. Rushworth, and T. Venema, 'TOSI: a tree oriented string interpreter for the design and implementation of semantics', _Software--Practice and Experience_, to appear.

4. W.F. Appelbe, _A Semantic Representation for Translation of High-Level Algorithmic Languages_, Ph.D. Thesis, Department of Computer Science, University of British Columbia, 1977.

5. L.J.M. Geurts and L.G.L.T. Meertens, 'Designing a beginners' programming language', Mathematisch Centrum, Amsterdam, 1976.

----

*One may be unhappy about this or one may rejoice that there is at least one environment where the individual is of primary importance.

6. M.S. Johnson, The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment, Ph.D. Thesis Draft, Department of Computer Science, University of British Columbia, 1977.

7. D.R. Ramer, Construction of LR(k) Parsers with Application to ALGOL 68, M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1973.

8. D.M. Ritchie, C Reference Manual, Bell Telephone Laboratories, Murray Hill, New Jersey, 1974.

9. T.B. Rushworth, T. Venema, and H.D. Abramson, 'TOSI', Proceedings of the 1975 International Conference on ALGOL 68, Stillwater, Oklahoma, 293-305 (1975).

10. T.B. Rushworth, Macros as a Method for Specifying Semantics, M.Sc. Thesis Draft, Department of Computer Science, University of British Columbia, 1977.

11. T. Venema, A Lexical Scanner Generator for a Modular Compiler Generation System, M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1975.

12. T. Venema, 'A lexical scanner generator for a modular compiler generation system', Proceedings of the Canadian Computer Conference, Montreal, 373-386 (1976).

13. T. Venema, TRUST User's Guide, Technical Manual, Department of Computer Science, University of British Columbia, 1976.

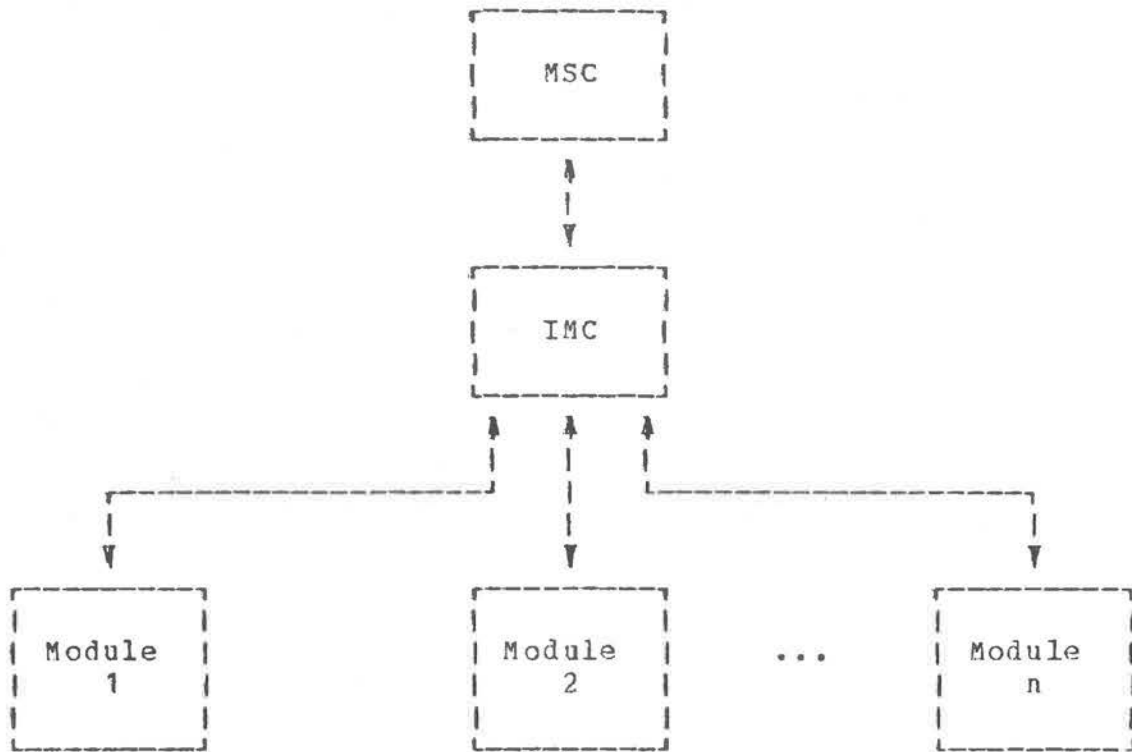

Figure 1. Basic Translator Structure