AN AUTOMATIC CODE SYNTHESIZER

W.K. Reinholtz
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California91 109

Abstract

This paper describes the design and architecture of an automatic code synthesizer we call the ACG. The input to the ACG is a list of facts that must hold for the generated code, along with domain-specific knowledge (design rules and patterns). The facts are writ ten by the application programmer. 'l'he design rules arc provided by the system architects. The ACG derives and emits code, currently C++, using an expert system to perform the reification. The ACG is in daily USC, supporting development of m ission-critical software. It produces about 40% of a 150KLOC product, replacing approximately onc staff programmer position,

1 NTRODUCTION

Background

This paper describes an automatic code synthesizer called the ACG. The ACG is part of an ongoing effort within the ZIPSIM project at JPL to increase programmer productivity and software product reliability.

The charter of the ZIPSIM project is to create high-performance spacecraft emulators. A ZIPSI M emulator consists of a number of component models, each with a full-fidelit y interf ace and arbitrary code (usually C++) to implement the interface. The models are put into the ZIPSIM framework, which provides mul t i-processor scheduling, runt imc model creation, interconnection establishment and breakdown, graphical and line-oriented user interfaces, and so on. The usual purpose of a ZIPSIM emulator is to execute unmodified flight software binaries on the emulator such that the software behaves just as it would on flight hardware.

Motivation

'1 he goal of the work wc describe here is to increase programmer productivity in two ways:

- The programmer need only provide a concise list of facts that describe the needed code, rather than write the code itself; and
- The structure of the facts is designed to al low a nun)ber of possible implementations, so that significant redesign of the product may be accomplished by al tering only the design roles, rather than manually altering all code impacted by the redesign.

The programmer provides a concise list of facts that apply 10 the code that is going to be generated. The tool then uses those facts and a Jwc-existing set of design roles to write the code for the programmer. Productivity is increased because (a) the programmer only writ es facts about the code to be generated, rather than the code itself; and (b) the programmer need not be intimately knowledgable of the design rules; and (c) the crier rate is lower and thus less time is required for bog fixing.

The programmer- provided facts arc as much as possible independent of the design rules u sed to create the output code. It is thus practical to implement a significant redesign simply by rewriting the design rules. The code can thus be evolved less expensively and more reliably, to meet changing requirements or to utilize improved technologies[1].

Related work

S ynthesis techniques can be applied to the whole. software life-cycle [Setliff93], outlined below.

- Requirement - A description of the dcsi red behavior is used to create a specification;
- Design - A software design is synthesized from a specification;
- Implementation - Code is synthesized from a design;
- Verification - Test cases are generated automatically;
- Maintenance - Maintenance is cased because design rules arc captured.

Onr work spans all but the verification phase described above, though the different i at ion bet ween phases is different: The ACG uses facts about desired code and design roles as input, and emits an implementation. The facts describe attributes that the code must have, but do not specify behavior or design: those arc captured in the design rules, so they may be easily altered.

Yellin and Strom [Yellin93] describe a synthesizer that creates adapt ors bet ween components that arc functional ly compatible but are not type compatible. Our ACG provides this capability, but it can also be configured to cause type compatibility via inheritance instead of adaptors, should that be desired. in either case, the facts and code writ ten by tbc programmer need not be modified.

Synthesis should not be confused with automatic code

1. We have used this capability several times, and it worked as advertised.

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

generat ion, which translates a graphical or textual higher-level language. into lower-lcvc] code. This was a popular topic some years ago. See e.g. [Razdow 1982,] as an example.

The history of the ACG

'J he ACG was implemented because wc observed that a significant amount of the code in a previous version of our product, though complicated, was essentially pro-forma once the design rules were understood. Unfortunately the design rules were (and still arc) sufficiently complicated that it often took days of struggling to learn thcm wc]] enough to appl y thcm [1], and the application itself was tedious and error-prone. Worse yet, each programmer went though the cycle a number of times, because each soft ware component created by the programmer required another application of the design rules.

Our initial goal was to write a set of development procedures that embod ied the design rules, so that the programmer would bc aided in the correct application of the design roles. Our intent was that the programmer would bc guided towards the creat ion of a set of facts about tbc code to bc writ ten, and then would usc the development procedures (structured as a series of IF . . . THEN . . . ELSE . . . ENDIF rules in procedural handbook st ylc [Wieringa92]), in conjunction with the facts to write the code.

It soon bccame clear that the procedures themselves would bc quite extensive and complicated, so wc considered writing a tool that developed customized procedures. The tool was to take as input certain facts about the code to bc written, automatically determine which procedures were applicable, and finally emit a customized procedure for the programmer to follow. Wc discussed the use of an expert system as tbc basis of this automatic procedure synthesis tool.

Finally (a fcw days later), wc observed that if wc could automatically derive the procedures, perhaps we could automatically apply thcm as well. The answer to this question turned out to bc "yes". and the ACG as described here was born.

Outline of the paper

The following section describes the software product to which the. ACG was applied. Wc nex t provide an overview of the operat ion of the ACG, then we exam inc it's input and output products in some detail. Next wc discuss the CLIPS expert system that we used as the basis of the ACG inference engine, and the architecture of our CLIPS rule-set. 1 'i n al I y, wc present some measu rements as to how much of the total body of code is actually gene.rated by the ACG.

1. Programmers more than once referred to the process as "painful".

ZIPSIM Architecture

'1 'his section describes those ZIPSIM architectural details that should bc understood in order to case reading of this paper.

ZIPSIM consists of a number of *components*, connected via strongly-t yped sp/ices. 1 'ach component as a"1 c1 [Oster93] interpreter wrapped in a class called *SimTcl*. Interpreters communicate. with amongst themselves via a connectionless R1'C-like protocol mechanism, embodied in a Tcl command called *simsend*.

- Component - A component is an object. augmented with certain mandatory interfaces that support component creation, registration, communication, and deletion. If a component has certain optional interfaces that connect it with the emulation scheduler, it is also a model. Components are named and registered in a central database.
- Splice - A splice is a high-performance component interconnection. Unlike simsend, splices are point-to-point and connection-oriented. Splices arc strong] y-t ypcd, so syntactic correctness of connect ions is enforced by the ZIPSIM framework. 'Each splice as a master, which originates i/o requests, and a slave, which responds to the requests. In RPC notation, the master is the client, and the slave is the server.
- SimTcl - SimTcl is a C++ wrapper wc place around Tcl that provides an interface more compat ible with C++ development, ant] is also augmented to make Tcl safe in a multi-threaded environment. 'Each instance of SimTcl has a name and is registered in a central database.
- simsend - Simsend is used to execute a Tcl command on a named SimTcl object. It may bc used by any SimTcl to execute a command upon any other SimTcl. It has lower performance than a splice interconnection, but since it does not require splice establishment machinery, it is more convenient to use and is appropriate where performance is not critical. Note that simsend is not the Tcl/Tk "send" command. The latter has subst ant ially lower performance but can commun icate bet ween Tcl interpreters that arc execu t ing on different machines. Simsend dots not operate across the network .

ACG OVERVIEW

The ACG consists of two software components: a specification compiler, which translates specifications into facts for the inference engine; and the inference engine, which synthesizes C++ code from the facts and roles.

The specification compiler takes as input a file containing human-readable facts about the component being built, and emits facts formatted as input for the inference engine. No inference is performed at this point: the contents of the input file d i rect l y imp] y the contents of the output file. We avoid inference in this phase so that the system design rules do not become scattered amongst the various ACG subsystems.

'1 'he inference engine. takes as input the output of the specification compiler, and produces C++ source code. The per-component facts arc combined with system-wide facts and rules, using CLIPS [Riley91] expert system tool to perform the inferences.

The following steps summarize a complete ACG usage Cycle:

- 1. The developer puts component specifications into <file>.nspec.
- * 2.< file> .nspccis proccsscdby factgcn.sh,andproduces <fi le>.nfacts. Note this translation does not use any input other than <file>.nspcc.
- 3. The inference engine, filegen.sh, processes <file>.nspec and generates a number of C++ files. The processing ma y cause a number of other nspec files to be read to provide tbc information required to generate the C++ code.

ACG INPUT

'J 'he specificat ions are structured as a comprom isc bet ween ease of input and interpretation on humans, and ease of processing by software tools. Our first inclination was to use a notation that did not imply any particular processing language. It soon became clear that if we were to offer a powerful input not at ion, we'd either have to invent our own language, or use an existing one.

We selected Tcl as our specification input language, because (a) it's adequate; and (2) wc already depend on it and have some ski ll in its application [1]. Some developers have used this to their advantage, by writing shorthand specification items, which expand into perhaps several actual specification items.

1 'ach specificat ion item has a comment field, which is inserted into the fact database along with the fact itself, The comment is used for several purposes. For certain dat a items (e.g. C++ function and variable declarations and definitions) it becomes the C++ comment associated with the item in the output C++ code. 1 'or others, it forms the basis of a help-text system. Finally, all such comments arc fodder for automat ic document generation[2].

'I 'here arc many specificat ion items that maybe (and some that must bc) used. The following lists some of the more significant ones., and what code is generated by the items.

- The name of the component - The name is used in many places, inducting the C++ class declaration, function definitions, and registration of individual instances of the component type.
- Member functions and variables - The ACG must create the C++ class declaration, as it controls the architecture up to and beyond the basic class inheritance structure. As such, all member data and member functions must be specified to the ACG. That is the primary purpose of th is specificat ion item. A side effect of this item is that the ACG can be used to create documentat ion describing each member funct ion and data item, which has proven useful.
- Splice t ypcs and names - Components may include splice capability. This specification item specifics the various splice types to be implemented by the componcnt, ant] the name of each port of each type. This is used to create a large amount of interface code.
- Simset - Simset is a Tcl command that provides access to component C++ variables. The ACG can generate all code to do so for most data types. Some developer assistance is required for structured data items.

ACG Input examples

The following example. shows a data specification item of low complexit y:

```
c {
No n-blocking SimTcl lock .
I f SimTcl isn' t 1 ocked, lock it
and ret.urn zero. ] f i t is 1 ocked,
ret urn non-zero .
} fdecl] ockpt henlock publ i c stat i c:
"i nt FUNC" "void" "voi cl" normal
not virt ua]
```

The first word following the comment list is always the type of the specification item being declared, in this case a function. The following parameters describe the item in detail sufficient to allow generation of the. necessary C++ code.

The above example is fairly simple, in that the translation

1. We have not regretted this decision. In fact it still appears to have been the right thing to do.

2. Wc have noted that programmers often initially neglect to fill in the comment field, then must go back later and add the comment so as to allow use of e.g. automatic document generation.

is somewhat obvious and trivial, Consider, however, the following examples:

```
c {The CPU regi st ers  qsimset R rw
uint16 o 31
c {The ] C} qsimset1  c {R(IC)} rw
uint16
```

1 'ach of these cause the creatio ı of a significant amount of code.'1 'be. first causes a simset function to be create.cl in the model (in this case a CPU) that provides Tcl access to the register away RIO..31], with read ant] write capability. The second example aliases a particular register (the IC, instruction counter), so the user does not need to remember it's numeric index'.

Each of these lines causes perhaps a dozen lines of C++ to be generated. There arc hundreds of variables that can be accessed in this manner. By using the ACG to create the interface code, wc can ensure uniform range checking, input type checking, implementation style, and so on. Otherwise each programmer would be rcqu ired to do each of these things independently, for each variable. Each instance of such code would provide another opportunity for a mistake to be made,

By using tbc ACG to generate this code, we gain in two ways: (J) if a dozen or so lines of ACG code arc correct, then hundreds of lines of C++ code arc correct; and (2) if wc decide to i mplement simset differentl y, we need only change a fcw lines of ACG code, rather than hundreds of lines of C++ code.

ACG OUTPUT

Output products

The ACG emits a number of C++ header ("h") and code (".C") files for each component. The name of each is mechanicall y derived from the name of the component ant] the purpose of tbc contents of the individual file. 'l'he following enumerates the types of files written by the ACG.

- Class declaration. This file must be automatically generated, because some of the coding under the control of the ACG requires that class members be declared and that certain parent (s) arc specified.
- Component creation and deletion. This code, much of it boi lerplate, is requ ired to support our component arch itecture. The code that registers components upon creation and unregisters upon deletion is boilerplate. The code to initialize the component is not boilerplate, as it may involve arbitrary code fragments,
- SimTcl creat ion ant] deleti on. Each component has a Tcl interpreter, wrapped in a class called SimTcl. This file contains the code that handles much of the. bookkeeping required of this design, It also contains code that, for each component class, creates a static SimTcl associated with that class that's used to spawn instances of the component class in the "exemplar" style of e.g. Self [Ungar87][2].
- Splice connection establishment.
- installation of Tcl commands. It is tbc nature of Tcl, and so SimTcl, that commands that arc written in C++ arc c1 ynam ical l y registered with the interpreter (as opposed e.g.to a conlJ~ilc-time lookup table). This file contains the code that registers all Tcl commands. It also contains code that, for each command, registers the help-text Of the command with the help subsystem,
- Automatically y-generated Tcl commands. Sonic Tcl com mands, e.g. those t hat are used for component creation and deletion and those that arc used for splice cst ablishment and breakdown, are automat ical l y defined as well as automatically declared. This file contains such defi nit ions.

Output techniques

The bulk of the code generated by the ACG is created using onc of four techniques, enumerated here.

- Direct code creation - This is the most powerful technique. An algorithm within CLIPS is used to construct the code, using a combination of procedural and rule-bascc] techniques. This method is reserved for applicat ions where the requ i reel code does not exhibit a structure that may be exploited by simpler techniques.
- #define/#include - The ACG writes a file. that uscs the C++ preprocessor "#define" to define certain macros, then uses "#include" to include a file that becomes customized according to the defined values. This technique is quite useful when the desired code product is sufficientl y regular, because the ACG need only cm it the necessary parameters, and because certain changes may be implemented by modifying only the include file, rather than making another ACG pass.

J. Wc have considered causing the act of declaring a variable to also cause its simset specification to be asserted, so all variables would be automatically accessible via Tcl. We have not yet done so, for Jack of time.

2. This is how wc avoid explicit central registration of all component classes. The static SimTcl registers itself upon creation, thus automatically creating a list of al lcomponent classes.

- Macro substitution - The ACG has a simple macro substitution capability, which takes as input a template file, and emits the template with certain tokens within the template replaced by ACG-specified values. This is an early ACG feature and has for the most part been replaced by the #define/#inchldc mechanism, because the latter is more powerful by virtue of its conditional constructs (#if . . . #endif) and better-controlled recursive macro evaluation, and the former offered no significant advantages with respect to the latter.

- String substitution - This method is similar to macro substitution, but rather than using an input file and an auxiliary macro substitution program, it is internally implemented and operates on strings. We often use this to generate code fragments.

ACG INFERENCE ENGINE

We used the CLIPS[Riley91] forward-chaining expert system development tool, developed by NASA, to implement the ACG inference engine. We first briefly describe CLIPS, then we describe our use of CLIPS, and finally some coding guidelines that we found useful.

Brief description of CLIPS

Clips has been described elsewhere in the literature. We thns only briefly, informally, and incompletely describe it to aid the reader in understanding this paper.

CLIPS takes as input facts, and rules. The CLIPS inference engine repeatedly matches the facts to the rule predicates, and for each rule that fully matches, the rule is scheduled for evaluation. The evaluation among other things may came other facts to be asserted and/or deleted, modification of the matching facts, values to be assigned to global variables, and i/o to be performed.

A fact is an ordered list of values. The first value is the type of the fact, and the remaining values together represent the value of the fact. Each position in the list may be given a name, thus providing something of a traditional record data structure.

A rule has a predicate (in CLIPS notation, the "left hand side,", or 1,11 S), actions (the "right hand side", or RHS), and salience. When the predicate of a rule is satisfied by some set of facts, the rule is scheduled for evaluation, which will cause the actions of the rule to be executed. Salience determines in part the order in which rules scheduled for execution are executed: The greater the salience, the earlier the execution, all other things being equal.

In essence, CLIPS repeatedly identifies a rule that has a predicate that can be satisfied by existing facts, then executes the actions associated with the rule. If more than one rule is eligible. for execution, a conflict resolution mechanism is used, which selects a rule by one of several prioritization schemes.

The design of the ACG ruleset

The ACG inference engine excretes in three distinct phases, controlled by rule salience: reification; code emission; and output file iteration. The reification is done once for a given set of output products. Code emission and file iteration occur for each output file.

Reification consists of inferring all facts that are implied by the initial facts and given rules, It is done first, and once, so that code emission and file iteration (both irreversible) do not have to be back (racked. If we allowed facts to be asserted during code emission, and any of those facts could imply changes to previously-emitted code, backpacking would be necessary. We have no mechanism for such backtracking, and imagine it would be quite difficult to implement, so we instead avoid the problem.

We used the CLIPS ndc-based notation to express most of the code emission algorithms, so some emission rules assert facts that control other emission rules[1]. Such facts have names that make it clear that they are only to be used for that specialized purpose, and by convention each is restricted to appearing in a very few rules (typically only one rule asserts such a fact, and only rule uses the fact in it's predicate). Our goal was to allow the use of rule-based algorithms, yet to ensure that rules asserted during the course of code emission could not have impacted previously-emitted code.

We would have preferred to avoid ANY modification of the fact database during code emission. It would have then become possible to implement an automated check for fact modification during code emission. However, CLIPS is an expert system, so its programming style tends towards the rule-based[2]. It would have been difficult to implement the code emission algorithms in the CLIPS procedural notation, and the. code would have been obscure as compared to the rule-based notation,

After some painful debugging experiences, we started routinely tagging each fact with what amounts to it's genealogy. We found this to greatly ease the problem of understanding exactly how the emitted code came to be as it was. For example, if a C++ function declaration fact is automatically generated, we put into the function comment field, the name of the rule that created the fact. If a

---

1. The astute reader will note we thus lied in the previous paragraph re: facts asserted first, and once.

2. CLIPS rules have a lisp-like appearance, but CLIPS does not provide anywhere near full lisp functionality. We would have preferred it otherwise.

later rule were to e.g. add another formal argument to the function declaration, the name of that rule would be appended to the list of rules that modified the fact. In this way a simple (and, wc have found, quite useful) execution trace is provided].

A preliminary version of the ACG did not include the file iteration phase. Instead, after facts were elaborated, it attempted to write all output files at the same time. This did not scale well, because both adding capabilities to the ACG, and adding components under the control of the ACG, caused the number of output files to increase. Wc soon reached a point where the operating system could not support the number of simultaneously opened files that the ACG required. We then implemented a file iteration scheme, whereby only a few files are opened at any one time.

We were concerned that it would be difficult to argue the. correctness of the ACG. 'J 'bough wc conjectured that most errors would cause subsequent compilation or linking failures and thus could not threaten the quality of the executable code, they certainly could prove difficult to find and fix. We were thus motivated to structure rules such that errors were likely to be avoided, and if not, so they could be easily identified and fixed.

COST ANALYSIS

The ACG is shown to have saved about 2(J work-months, worth about $300K. The savings is increasing with time. The total investment in the ACG was about two work-months.

Cost to build the ACG

The first version of the ACG was created by the author over a period of about two weeks in carl y 1994. After some months of usage, it was completely rewritten in November 1994 by the author in another three weeks, resulting in the system described in this paper. Minor improvements and additions have been made since then. The total investment, including subsequent maintenance, is about two work-months.

Cost savings from use of the ACG

As of early December 1995, ZIPSIM consists of a total of 147K1 .OC of code, thus:

53K1.OC automatically generated by the ACG.

74KI .OC manually written.

20K1.OC recycled.

1. In retrospect wc should have made such trace information an explicit field of each fact, rather than use the. ad hoc method presented here.

Ignore the recycled code and assume that all code that is now automatically generated would have otherwise been manually written at roughly the same cost as the other manually-written code.

74K1 .OC of ZIPSIM has been manually written, in about 50 work-months, thus productivity was about J .5KLOC/work month. The ACG accounts for 53K1.OC of code, or about 35 work-months at that same productivity. '1'here are about J OK I .OC of specifications. Assume it's twice as hard to write a specification Jinc as it is a C-t + code line. The specifications thus took about J 3 work-months to create. Since it cost only 2 work-months to create the ACG, it has saved about 20 work-months, worth about $33(JK. Basically, it replaced at least one programmer.

Savings analysis

This section examines the strength of the savings argument made above, and provides an intuitive basis from which the marginal utility of the ACG may be evaluated.

The ACG emits 53KI.OC of code based on 10KI .OC of specifications. The code is specifically intended to appear more or less as it would if manually written. For the sake of argument, let's assume a Jinc of specification is twice as difficult to write as a line of code. The ACG thus effectively replaces (53-2*10=33) 33KI.OC of code.

Recalling that there are 74KI .OC of manually-written code, that means that for each manual K1 .OC there is (33/74) 0.45KI.OC of effective automatic code (that is to say, we've already adjusted for the cost of the specification that was used to create the code).'1 'hus if there were no ACG, each programmer would have to write about 1.5 lines of C++ for each line now written, That implies that for every two or three programmers now on staff, there would have to be another one hired to write the code now produced by the ACG. Given that it only took a couple of months to create the. ACG, the savings claim seems reasonable.

In order for the ACG to have no marginal advantage, it is necessary that the effort to write the specifications is at least as great as the effort to write the code that is synthesized from the specifications. As there are 10Klines of specification, and 54 Klines of code, that means that the specification would have to be over five times more difficult to write than the code. Our experience is that this is far from true. In fact each line of specification is probably about as easy to write as a line of code.

SUMMARY

We have presented the motivation for and design of a code synthesizer that is in daily use. It produces over a third of the new code in the product, and has proven highly cost-effective, saving about $300K so far and providing more savings every day.

ACKNOWLEDGEMENTS

The concept of a tool to automatically create procedural checklists was born in April 1993, at Jims Burgers in Altadena, California, during a lunch-time conversation between the author and Dr. George Luger of the University of New Mexico. Subsequent conversations with Bill Robison of JPL led to the concept ofs ynthesizing code, rather than procedure checklists. The members of the ZIPSIM development team provided many ideas as to what code could be effectively synthesized.

REFERENCES

Oster93 — J.K. Osterhout, *An Introduction to Tcl and Tk*, AcJdison-Wesley, 1993.

Razdow82 — A. Razdow, R. Hackler, and R. Smaby, "Automatic code generation steps up productivity", *Electronic Design*, pp. 163-167, December 1982.

Riley91 — G. Riley, "CLIPS: An Expert System Building Tool", *Proceedings of the Technology 2001 Conference*, NASA, 1991.

Setliff93 — D.E. Setliff, "Knowledge Representation and Reasoning in a Software Synthesis Architecture", *IEEE Transactions on Software Engineering*, vol.18, no.6, pp. 523-533, June 1992.

Ungar87 — D. Ungar and R.B. Smith, "Self The Power of Simplicity", SIGPLAN notices, vol. 22, no. 12, December 1987.

Wieringa92 — D. Wieringaa, C. Moore, V. Barnes, *Procedure Writing principals and practices*, Battelle Press, 1992.

Yellin93 — D.M. Yellin and R.E. Strom, "Interfaces, Protocols. and the Semi-Automatic Construction of Software Adaptors", *Proceedings of OOPSLA 1994*, pp. 176-190.