8-1-2008

Automated Code Generation for Industrial-Strength Systems

Thomas Weigert
Missouri University of Science and Technology

Frank Weil

Aswin van den Berg

Paul Dietz

et. al. For a complete list of authors, see https://scholarsmine.mst.edu/comsci_facwork/237

Recommended Citation

T. Weigert et al., "Automated Code Generation for Industrial-Strength Systems," *Proceedings of the 32nd Annual IEEE International Computer Software and Applications, 2008. COMPSAC '08*, Institute of Electrical and Electronics Engineers (IEEE), Aug 2008.
The definitive version is available at https://doi.org/10.1109/COMPSAC.2008.26

Automated Code Generation for Industrial-Strength Systems

Thomas Weigert

Missouri Univ. of Science & Technology
Rolla, Missouri

Frank Weil, Aswin van den Berg, Paul Dietz, and Kevin Marth
Motorola
Schaumburg, Illinois

Abstract

Model-driven engineering proposes to develop software systems by first creating an executable model of the system design and then transforming this model into an implementation. This paper discusses the design of an automatic code generation system that transforms such models into product implementations for highly reliable, industrial-strength systems. It provides insights, practical considerations, and lessons learned when developing code generators for applications that must conform to the constraints imposed by real-world, high-performance systems. Automatic code generation has played a large part in dramatically increasing both the quality and the reliability of software for these systems.

1. Introduction

Model-Driven Engineering (MDE) [1] focuses on a development process that makes models the central development artifacts. The final software that is delivered is derived from these models and is never directly edited. Tools are able to transform design models into implementations in target languages. Nevertheless, commercial software development has not been able to fully capitalize on these advances yet. The obstacles proved to be the code generators. While impressive results were demonstrated in non-constrained domains or with toy examples, automatic code generation has proven elusive in many application areas, in particular, for embedded, real-time systems.

In this paper, we discuss our experience in developing commercial-strength code generators that have proven effective in highly constrained domains, in particular for the development of telecommunications applications. We examine capabilities an automated code generation system must possess for it to effectively support these domains.

We rely on the transformational implementation approach to software development that uses automated support to apply a series of transformations which change a design model into an implementation. Transformational implementation enables the rapid realization of the model and the maintenance of the product at the level of the model.

We begin by summarizing constraints on program transformation systems that are implied by the constraints under which industrial software products are developed. Many of the difficulties in realizing a high-performance code generation system are due to these constraints. In order to support the deployment of MDE at Motorola, we have implemented the Mousetrap program transformation system. This system accepts high-level models in domain-specific notations geared at the development of real-time embedded systems in the telecommunications domain: UML [2], SDL [3], and PDU definition languages [4]. It generates high-performance product code from these models. This system has been in use in Motorola product organizations for roughly ten years, and a significant portion of application code for Motorola telecommunication network elements has been generated by Mousetrap.

In Sections 2 and 3, we provide context and motivation for this work based on results obtained for the telecommunications domain, and we describe the constraints implied by that domain. In Section 4, we give an overview of the Mousetrap program transformation system, compare different types of transformation rules, and describe the architecture of the transformation process realizing high-level models in C target code. Finally, in Section 5, we describe the transformation process itself. This paper summarizes our experience and does not attempt to compare our approach to other realizations of automated code generation.

2. Motivation and Benefits

Automated code generation (ACG) and MDE hold much promise, including reduced defects and increased productivity. Achieving these benefits in a commercial environment, however, can be quite difficult when one takes into account the various mitigating factors such as budgets, time-to-market, legacy code, resistance to change, etc. Motorola has implemented a development process based on MDE including ACG, and has achieved the expected benefits. While it is impossible to completely separate the benefits of MDE from those of ACG alone, our experience shows that essentially no benefits were gained *until* ACG was applied: We found that in spite of demonstrated significant quality benefits, many development teams resisted the use of MDE since they felt that additional effort required to make their design models operationally interpretable amounted to having to "code twice", first at the level of the design, and then when they wrote the code based on that design. In our experience, users will only create the lowest level artifact to which automation can be applied. This section describes the overall benefits observed. Additional detail can be found in [5].

Much of the up-front benefits of ACG come from automation of the labor-intensive and error-prone coding tasks. Additional benefits come from the separation of platform concerns from product concerns. That is, development engineers can focus on the requirements of their product, and they can leave the platform details to experts in that area.

Pushing much of the development detail into the code generator allows designs to be more abstract, which results

0730-3157/08 $25.00 © 2008 IEEE
DOI 10.1109/COMPSAC.2008.26

in models that are easier to produce and maintain. For example, we have found that fewer inspections are required to ensure the quality of the developed code than when using hand-written code. On average, developers rely on three inspection cycles applied to the model instead of four applied to the code. In addition, inspection efficiency is higher, having increased from 100 source lines per hour to between 300 and 1000 source lines per hour. Thus, not only are fewer inspections required, but also the remaining inspections are much more efficient.

Productivity improvements in the development of features on a core network element average 5X to 8X with MDE based on ACG. The productivity improvement during the development of typical protocol data marshaling code is about 16X. The effort spent in the design phase increases, but this is more than made up by the dramatic reduction in coding effort.

ACG has also resulted in significant quality improvements. Overall for MDE and ACG, metrics data shows that there is approximately a 7X reduction in defect density. In addition, defects are found earlier in the life cycle (roughly double the rate of defects are found in the design phase following the MDE approach) and no defects are introduced in the coding phase. As the cost of fixing defects increases exponentially with the distance between where a defect is introduced and where it is discovered, this translates into large cost savings.

In addition, there are several back-end benefits. For example, generated code does not become scattered with patches. When enhancements or defect fixes are made, the code is regenerated (and re-optimized) from scratch. Also, a fix made in the code generator is universally applied, so overlooked fixes do not occur. Our data shows that because of this, defect-fix cycles have been reduced from 25-70 days to 24 hours in some cases.

The lack of platform detail embedded in a design means that reuse of designs and tests between platforms or releases is enabled. This yields greater savings than reuse by copy-and-paste. For example, we moved a base site controller application from a rack of MC6809 cards with distributed memory to a shared-memory computer. All changes required to the software architecture were performed by the code generator. In another example, an application was migrated to a lower-cost platform. The effort was 10 staff days to capture these differences in the code generator as compared to an estimated 80 days for hand porting the code.

While the cost of creating and maintaining a code generation system are high, the applicability and overall savings possible make the effort well worthwhile. For example, some of our network element teams have used ACG for 70% to 80% of their application code. ACG rates are lower for systems with a large amount of legacy code.

We have analyzed the source code for each of the components of a telecommunications system release and categorized the modules of source code by the design elements from which they are typically derived: (i) Code that is specified by state machines, (ii) code that is highly algorithmic or manipulates data, (iii) code that is low level and not captured in designs, and (iv) code that is described by other means, such as GUI layout or database design tools.

The distribution of the four categories varies between network elements. Our experience is that all of the state-machine oriented code can be derived from designs, as can most or all of the algorithmic code. The low-level code is unlikely to be generated automatically. Use of ACG in the "other" category varies, but this code comprises a relatively small percentage of the overall application. Rolling up the data for various telecommunication systems reveals that the potential for MDE and ACG is between 75% and 95%. The individual percentages are less important than the message: a significant portion of a telecommunication system is amenable to ACG from high-level designs.

3. Domain Constraints

Current ACG techniques often either do not scale to the size needed for real applications or generate code that does not meet the constraints of the environment in which the applications are embedded. What differentiates the theory of program transformation from its practical application is the set of constraints under which a real system must be developed. There are several characteristics of industrial-strength systems that are applicable to ACG:

System size. Most applications on a telecommunications network element are composed of hundreds of thousand lines of code. Further decomposition typically comes with the penalty of decreased performance of the overall component due to increased context switching and interprocess communication.

Performance. Performance requirements relate to execution speed, memory use, throughput capacity, latency of responses, etc. Even with the increasing power of processors and the lowering cost of memory, these requirements are not being relaxed significantly. The amount of work that the applications are required to do is increasing at a pace often faster than the capabilities of the underlying processing hardware.

Platform Interaction. Systems have increasingly sophisticated interaction with other applications and the underlying target platform. Systems must also be "well behaved" on the target platform, e.g., being able to answer health probes in a timely manner, honoring and properly handling OS-level signals, using the dictated inter-process communication mechanisms and middleware layers, and linking with supplied library files.

Reliability. Applications operate in a hostile environment where malevolent users try to gain control of a system. Attempts at exploiting system weaknesses such as buffer overflow are common in every domain. In addition, there are often severe penalties for system failure, and field debugging must be facilitated. An infrastructure component often cannot fail for any reason without far-reaching consequences.

System Life. Typically, systems are long-lived, are part of product families, and are incrementally enhanced for successive releases. Rarely are systems developed in isolation and then never used again.

From the above characteristics, the following constraints can be derived both on the code that is generated and on the ACG system itself.

- Transformation algorithms must scale well.
- Transformation memory must be wisely managed.
- Overall transformation time must be minimized.
- The ACG system must be robust in the presence of

design problems and be able to provide useful feedback to the designer.
- It is very important to generate highly optimized code. Generated code rarely looks like hand-written code, so often compilers are not able to optimize that code well.
- For both performance and footprint, application memory must be well managed, with dynamic allocation kept at a minimum.
- Run-time interpretation must be kept at a minimum.
- Selection of concrete implementations of abstract data types is very important.
- With certain simplifying assumptions, additional optimizations can be made, but it requires the user to abide by the simplifying assumptions.
- The generated code must use a chosen platform API.
- System interfaces must be accommodated, including callback functions, asynchronous events, etc.
- It must be possible to trace the generated code back to the design.
- Security aspects outside of the scope of the design itself must be accommodated (e.g., prevention of buffer overflows, validation of input data, etc.).
- Reliability aspects outside of the scope of the design itself must be accommodated (e.g., recovery from memory allocation failures).
- Designs must be as free as possible from platform details since it is almost guaranteed that they will change as the application evolves over its life.

These constraints set the framework for why industrial-strength ACG systems are structured the way they are and why they perform the transformation steps that they do.

4. Automatic Code Generation

A design should be as free from platform considerations as possible. One would ideally like to target a given model to widely disparate platforms with no changes to the model itself. This tenet implies that considerations such as communication protocol implementations, error handling, timer and memory interfaces, and platform-generated events (such as interrupt signals) should be outside of the model itself. These considerations are, however, critical to the generation of correct and optimized code.

Figure 1 provides an overview of the ACG process, highlighting the additional information needed to generate the code. The cylinders in the figure represent manually created inputs to be transformed into code and information to be used during the transformation. The document symbols represent generated outputs from the transformation process. The thick arrows represent transformation processes, and the thin arrows represent items that are used as is.

As can be seen from Figure 1, the actual model is only one of several inputs necessary for ACG. The other inputs play an equally important role:

Compiler and platform specifics. Some aspects of target languages are left unspecified. For example, the C standard does not define the size of integers. The most effective strategy is to automatically generate types that can be used in the model, but there are other implications. For example, C compilers do not handle a literal of the most negative integer intuitively. If this literal is in the model, how it is

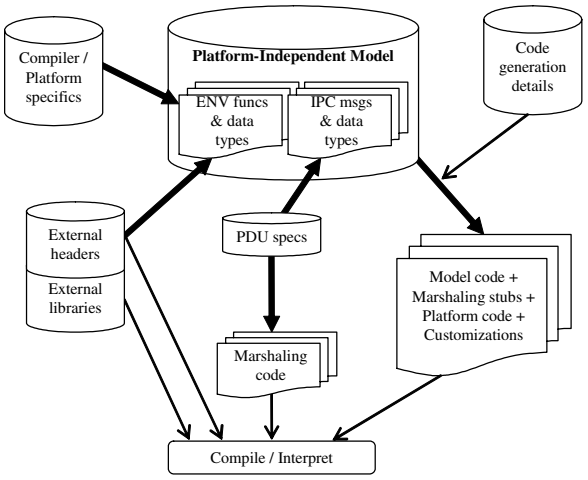

Figure 1: Automatic Code Generation Overview

translated to code will depend on the integer size on the target platform.

External headers. If functions or constants from outside of the model are used, the proper external declarations should be automatically generated. This includes not only the declarations themselves, but also the closure of all the related declarations.

External libraries. It is critical that the declarations used in the models be consistent so that the calls in the generated code exactly match the library functions.

PDU specifications describe the messages that are sent to and from the component as well as the data they carry. These specifications should be used to automatically generate both the messages and data types used in the model as well as the corresponding marshaling code for the target platform.

Code generation details describe the special instructions needed to generate efficient code that does not violate any platform constraints. Typical information declared here includes middleware APIs, details of the inter-process communication mechanism, details of error handling, and logging and debugging information that should be added to the resultant code.

4.1 Mousetrap

The Mousetrap program transformation system manipulates a source program by the successive application of transformation rules [6, 7]. A transformation rule describes a semantic relation between two program fragments, typically weakened versions of equivalence. For example, a correctness-preserving transformation rule states that two programs produce identical output given the same input, ignoring erroneous inputs, undefined situations due to nontermination, or the like.

The language used to express transformation rules consists of a means of defining grammars and abstract syntax trees, as well as forms for constructing, inspecting, matching, and traversing the trees. Mousetrap transformation rules are expressed in an extension of Common Lisp, the im-

plementation language of the transformation system. In this way, it is possible to intermingle rule-based and procedural forms to express program transformations. Mousetrap then applies these rules to an input program or program fragment and yields the parse tree of the resultant program.

A transformation rule has an optional variable declaration section, a pattern part, and a replacement part. Both pattern and replacement are descriptions of a class of parse trees (we will refer to these as terms). In their simplest form, terms are written as source text in either the source or target grammars. They consist of the name of the grammatical class of a production of the respective grammar and some source text enclosed in quotes. A term is constructed by parsing this source text as deriving from the specified grammatical class in the respective grammar. Source and target grammars are not necessarily the same.

A term may simply be a literal description of a parse tree, or it may be the description of a parse tree with some subtrees (or the whole tree) left unspecified but restricted to a given grammatical class. Subtrees are left unspecified through the use of variables standing for subtrees of a particular grammatical class.

Applying a transformation rule means replacing a program fragment by another program fragment: If the pattern matches a term of the input program, its variables are bound to the respective subterms matched and a replacement term is constructed from the replacement part of the rule and the bindings for the variables previously created. Then the resultant replacement term is inserted into the input program in place of the matched term. The pattern and replacement must both derive from the same grammatical class for this replacement to be meaningful. Terms are matched against patterns modulo equational theories that hold for the language in which the source text is parsed. Equational matching, in particular modulo a theory of lists, is often useful, but care must be taken in writing patterns to avoid non-scalable rules.

The Mousetrap system itself consists of several components: a rule compiler; a facility that produces parsers, term compilers, unparsers, and definitions from an EBNF notation; and an engine providing order-sorted, conditional term rewriting, term identities, and flexible evaluation strategies.

4.2 Transformation Rules

Transformation rules can be categorized by their general purpose:

Translation Rules replace constructs in one form or language with the equivalent construct in another form or language. Examples of these rules are "flattening" of inheritance features, inlining of packages, translation of state machines into nested loops, etc.

Canonicalization Rules render all uses of a construct into a single form. These steps minimize the number of rules that need to be written in other phases by reducing the number of forms that must be matched against. Different from translation rules, they do not help the progress of the overall transformation, but instead mitigate how complex it is to write the rules themselves. For example, `if` statements which do not have an `else` part are converted to normal form in which the `else` part is present but empty.

A more substantial canonicalization transformation is expression lifting. After this transformation, the program has the property that all expressions that have side effects are either the top-level expression of expression statements or the right-hand side of assignment statements. All more deeply nested expressions are side-effect free.

Information Rules collect information used by other rules, but do not in themselves transform the code. Examples of this type of rule are determination of the lexical scope of all identifiers, analyzing alias usage and expression side effects, or propagation of type information and variable range bounds.

Semantic Analysis Rules perform both static and dynamic semantic analysis on a program to look for potential defects. Examples of static semantic errors to be caught include `decision` statements without an answer part for all possible values (and where there is no `else` part) or `decision` statements in which there are overlapping answer parts. Examples of dynamic semantic errors are out-of-bounds indexes, dereferencing a null pointer, or assigning a value that is out of range to a variable.

Optimization Rules create code that is semantically equivalent but performs better in some aspect such as execution speed or memory utilization. Generally, these rules are applied until either the term stops changing or a fixed number of iterations of the optimizer have occurred. The order of transformations and the number of iterations of optimizations were worked out by experience to both maximize the efficiency of the generated code and minimize the space and time used by the transformations.

Examples of optimizations include those that are standard in compiler technology such as constant folding, function inlining, and common subexpression elimination. Other examples are more specific to higher-level model transformation such as state-machine simplification, compile-time garbage collection, and reduction of variable bounds.

An extensive group of optimizations involves simplification of expressions according to various algebraic laws. At this stage, canonicalization has ensured that no subexpression can have side effects. This fact enables a wide variety of algebraic laws to be used without regard for the order of occurrence of subexpressions. Expressions may also be eliminated entirely if this is algebraically allowed. Mousetrap has more than 400 transformations for expression simplification.

A set of optimizations that are particularly important involves the elimination of unnecessary copying of values. In a language with value semantics, such as SDL, large aggregate objects (records, arrays, sequences, etc.) are implicitly copied at each assignment, function call, or built-in operation. These copies can be very expensive, even changing the complexity of the algorithm being expressed if not removed.

Transformations must also recognize common idioms. For example, accessing elements of a list sequentially in a loop can be transformed into accesses using an auxiliary pointer that is moved down the list in step with the index variable.

4.3 Layered Approach

The transformation from the input language to the target language is performed in several stages (see Figure 2). This layered approach has two main benefits: it allows the modular addition of input and output languages, and it maximizes the reuse of the rule base by allowing the majority of the

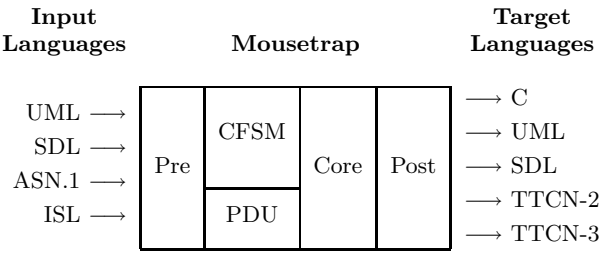

Figure 2: Transformation Process Overview.

transformation work to be done on one common language. Each stage of transformation has one or more languages associated with it. The translations between these languages are relatively simple syntactic changes and are independent of the main transformations within the stage.

The first transformation stage (Pre) is a preprocessing stage that converts an input language into a simpler and more uniform representation by using single forms to represent variations of constructs and by requiring or eliminating optional elements as appropriate.

The second transformation stage has two related parts: one that deals with the main behavior of a design (named CFSM for the concept of communicating finite-state machines) and one that deals with protocols (named PDU for protocol data units). This stage is responsible for understanding the semantics of the input languages. The main purpose of these rules is to transform the constructs that are specific to a given language or protocol into the set of common, or "core", programming features. It is at this stage where the domain-specific constructs of the input language are eliminated and realized in terms of the more generic Core constructs.

In practice, CFSM is a family of languages that share a common grammar. Transformation rules successively remove parts of the syntax while maintaining the same semantics. For example, the rules replace communication constructs such as channels, signal routes, gates, connectors, etc., with simple direct addressing; replace state machines with loops and `decision` statements; replace import/export with global variables, etc. Going from Core to C, the rules perform primarily syntactical transformations, such as replacing various loop forms with `do`, `while`, or `for` statements, or replacing `decision` statements with `case` statements or `if` statements.

Since the first two stages are most similar to the input language, most semantic checking is done in those stages. It is important not to assume that the input will be defect free. Transformations must be structured to be able to handle syntactically correct inputs that contain violations of both the static and dynamic semantics.

In addition, designers should be informed when constructs are encountered that are probably wrong because they cause transformations that would not likely be intended. For example, consider code that declares a variable to be in the range of 0 to 10, and then later checks if the value of that variable is greater than 10 (e.g., as a loop bound check). Transformation rules can easily detect that it will never be valid for the variable to be greater than 10, and that, therefore, the code is dead. It is unlikely, however, that the designer deliberately put in dead code, so an appropriate message should be generated. If this is not done, it may be very difficult to determine in the generated code why the expected code is not present, especially if there is a snowball effect where successively larger parts of the code are removed because the dead code causes subsequent parts of the code to become unreachable.

Care must also be taken to ensure that messages presented to the user actually represent language issues and are not artifacts of the transformation process. For example, when putting the code in canonical form, all `decision` statements are given a `default` branch. If a later optimization step removes empty branches, it would be ill-advised to present information to the user that the system removed a branch that the user has no knowledge of and no way of understanding if there is any practical effect.

The Core stage uses a domain-independent internal language as a common form for the multiple input languages. Core contains constructs relevant to the typical target languages generated. In particular, Core provides pointers (references to memory locations).

The majority of the transformational work is performed in Core. After optimizations are performed, a separate phase implements processes via functions operating on structures containing the part of the process state that may persist across the points where the process execution may block. This may sufficiently obfuscate the control flow of the process to the point that optimizations may no longer be possible. This is one reason that those optimizations are not left to the C compiler. The final stage of transformation performs post-processing in what is essentially the target language. These are relatively simple translations such as applying rules related to specific compilers. As a final step, the resulting term is pretty-printed in the surface syntax of the target language.

4.4 Platform-Specific Interface

Transformation rules are able to embody sufficient knowledge of the target platform so that the application engineers are required to specify only simple and concise choices that will parameterize the code generated for a specific platform. Among the capabilities that must be provided by the transformation system to implement the constructs and features used within a design model, either explicitly or implicitly, are the following:

Task / Thread / Process Management. Various operating systems treat tasks, threads, and processes disparately, and the code generator must tailor the generated platform interface to them. When the target is a general-purpose operating system such as Linux, an application is typically mapped to a process with multiple cooperating threads of execution. Other real-time operating systems may provide only cooperating tasks without the benefits of process-based memory protection or may provide a process abstraction without an additional fine-grained thread abstraction. These parameters are not reflected in the input language but are handled by the code generator.

Memory Management. The delegation of responsibility for memory management to the code generator contributes to the enhanced software quality associated with MDE. The memory management implemented by the code generator ul-

timately relies upon the platform memory services (e.g., the POSIX memory API), but efficient memory management may require that memory be allocated and partitioned into pools with specified numbers of fixed-size blocks at application startup. When appropriately configured using capacity requirements, memory pools ensure that an application will obtain the memory it requires as it executes regardless of the behavior of other applications on the same processor.

Timer Management. Applications may require thousands of concurrent timers, but the underlying platform may not be able to directly support that many active timers. Experience with timer management has suggested the compromise of employing a single platform timer to dispatch from an internal queue that manages the timers utilized within the design model. Thus, the number of timers that can be supported in the design model is constrained only by the size of the internal timer queue (if statically allocated) or the available memory (if the queue is dynamically allocated) and is not subject to potential limitations within the platform timer service itself that might be encountered if a one-to-one mapping of platform timers to model timers is assumed.

Interprocess Communication. Common platform middleware and operating systems support a formidable number of abstractions and mechanisms for interprocess communication (IPC), including TCP and UDP sockets, message queues, mailboxes, ports, remote procedure calls, signals, and callback-based message services involving registration and subscription. The platform interface must strive to enable maximally portable designs while accommodating designs in which IPC is exposed when necessary. When dynamic behavior is required within a design, a PDU may be augmented with an additional field dedicated to platform-specific detail, and the generated platform interface code will utilize the dedicated field appropriately.

5. Transformation Process

When transforming a design from the input language into the target, the transformations occur in stages:

- Traverse the input to generate an equivalent design in a simpler and more uniform syntax, simultaneously checking for some of the simple-to-find semantic defects and filling in derived information.
- Annotate the term with type and scoping information.
- Analyze the annotated term to detect semantic errors.
- Put the resulting design in canonic form.
- Optimize and transform state machines.
- Choose concrete representations of abstract data types.
- Translate the resulting term to Core.
- Perform canonicalization and optimization.
- Implement all data types.
- Perform further optimization on the canonic terms.
- Implement processes.
- Translate from Core to the target language.
- Perform final polishing of the resultant code.

5.1 Initial Translation

The first step is to construct a term that can be manipulated by the rule engine. In this transformation stage, as in every stage, the system must ensure that the transformed code conforms to the target language. This constraint implies that the rule system must respect the keywords, punctuation, and lexical structure of the target language. For example, if the input language has a variable named `for`, but that is a keyword in the target language, it must be renamed.

5.2 Annotations

Once the term has been constructed, it can be annotated with additional information that will aid in the further analysis and transformation. Two of the most useful annotations are the type of every expression and the scope path of every identifier. The scope annotations can be easily derived in a top-down traversal of the term by keeping a stack of the scoping units that are encountered during traversal. When an identifier is encountered, the associated name table entry is annotated with the full stack of scope units. The type annotations are more complicated to derive. The basic algorithm is to traverse the term, annotating expressions that are initially known, such as the types of unique literals and constants. From these known points, the surrounding subterms are annotated with their types and the process is repeated until there is nothing else to annotate.

At this point, it is also possible to insert runtime checks into the code for many of the potential semantic problems. Later optimization stages remove those checks that can be statically determined to be unnecessary. However, experience has shown that not only does this significantly increase the code size since a large portion of the checks cannot be removed, but there is also no single answer on what should be done when the check fails at runtime. It is typically more effective to report such problems back to the designer so that the underlying cause can be fixed.

5.3 Design-Level Optimizations

For code performance, it is important to minimize the amount of runtime processing. For code size, it is important to minimize the amount of supporting run-time infrastructure. An example that combines both of these involves the interprocess communication mechanism in SDL and UML. The semantics of a signal send require that the receiver of the signal be determined at runtime by following the path that the signal takes to its destination process. Not only does this resolution take extra time during execution, but also the entire resolution mechanism as well as the signal path mechanism must be represented in the code.

It is far better to statically determine the receiver at transformation time by analyzing the communication structure of the design. If it is not possible to statically determine the receiver, either the signal is being sent directly to a process using the unique handle of the process (in which case the receiver is a simple lookup in the process table at runtime) or the send is nondeterministic (which is rarely, if ever, what the requirements intended).

Other design-level optimizations are performed on state machines. For example, grouping common transitions together based on the number of signals involved can minimize the number of cases that need to be searched at runtime. It is also simple to detect the cases that can be further optimized. For example, if a state machine consists of only one state but that state has several inputs, then the state itself does not need to be explicitly represented. Alternatively, if there are multiple states but only one input signal,

then the infrastructure that matches on the signal type can be removed. Another case that can be optimized is a state machine with multiple states and multiple signals, but each state only handles one signal and no signal is handled in more than one state. In that case, it is only the sequencing of the signals that is relevant and the states themselves do not need to be transformed.

5.4 Abstract Data Types

The concrete implementations of abstract data types must be selected. General trade-offs can be made between minimizing the space required and minimizing execution time, and the final implementation choice must often be made by supplying additional information resulting from profiling the executing code. A viable scheme to handle the selection of the concrete implementation is to have several implementations available for each abstract data type. The specific implementation can be chosen based on heuristic rules related to the expected performance of the implementation.

For example, designs typically require the use of a data type that provides map (hash-table-like) functionality. Two additional pieces of information are needed to produce high-performance code: (i) Size bounds, as often the maximum number of mappings to be stored is significantly smaller than the size implied by the domain type. (ii) Mapping density, i.e., the maximum number of elements compared to how many there are typically. With this additional information, it can be determined how to implement the map (e.g., as a sparse array, a hash table, a regular array, etc.). Only knowing the domain and range types, the transformation system would be forced to choose a concrete data type that would allow a potentially very large number of tuples, such as a hash table created through dynamic memory allocation.

To help choose among candidate implementations, analysis determines which operations on the data type are used (or not used) in the model. For example, suppose that the transformation rules determine that the tuples of a map are never accessed directly through the domain values except for a single delete call, but instead the data type is frequently iterated over. This would indicate that a concrete implementation that favors iteration, such as a linked list, would be preferable over one that requires extra steps to determine the next value.

5.5 Translation to Core

As the final stage in the process of translating from the PDU/CFSM stage to Core, the term is transformed in a single pass that changes the types of the nodes in the term into their direct equivalents in Core.

5.6 Canonicalization in Core

Core is a procedural language for which ease of transformation was a more important goal than ease of programming. This led to a preference for uniform syntax over syntactic sugar. For example, instead of providing an `if` statement, conditional execution is always expressed using a `case` statement.

An important aspect of Core is that it assumes that integers have an arbitrarily large but finite range. This range information is part of the integer type. The types of integer expressions, then, are typically different from the types of the associated subterms. For example, if the variables `x` and `y` have types `int[0,10]` and `int[-100,200]`, then the expression `x + y` has the type `int[-100,210]`, i.e., it can take values between -100 and 210. Unlike C, arithmetic operations have the normal mathematical meaning in Core—semantics dependent on word size, such as rounding and overflow, do not exist in Core. As a result, it is easier to write correctness-preserving transformations as well as to reason about the types of expressions in a target-independent way.

Additionally constraining Core terms simplifies many of the transformations. Terms that satisfy these constraints are called "canonic". Examples of these constraints include:

- All expressions are well-typed.
- All names not externally visible are distinct.
- All control flow through procedures must end in a `return` statement, even for procedures that do not return values.
- Named constants do not appear in expressions.
- The only expressions that have side effects are either expression statements or the top-level expression in the right-hand side of an assignment statement.

The last constraint is important for at least two reasons. First, C does not define a standard evaluation order for expressions in most cases. If one is translating a language such as SDL that does have an order defined, it must be enforced by the introduction of temporary variables.

Second, many subsequent optimizations may be more easily expressed if most expressions are guaranteed to be without side effects. Many expression optimizations in Mousetrap either cause expressions to be moved with respect to one another or cause the number of occurrences of the expression to change.

Enforcing this constraint while preserving the semantics of a program requires that all subexpressions be "lifted" (replaced with a temporary variable that is assigned in a preceding statement) if the subexpression has side effects, as well as any subexpression to their left whose value may be changed by the lifted expressions. This is achieved by annotating each expression with information on the set of "abstract locations" that it may access and the set that it may modify. With this information, it can be conservatively determined if one expression can commute with another.

5.7 Optimizations

A large set of transformations in the Mousetrap system is devoted to optimization.

One of the most complicated and expensive optimizations is forward analysis. In this optimization, assertions about the values of variables and other expressions are propagated forward along paths of possible control flow. The optimization makes use of side-effect information (attached as attributes to statements) to remove assertions as they potentially become invalidated. Propagated information includes:

- That a variable or expression has a constant value (constant propagation).
- That a variable or expression is equal to another variable or expression, and this was the result of assignment of the second to the first (copy propagation).
- That a pointer is not null.
- That an object with nontrivial constructors is in an initialized ("nullified") state.

– That the value of an integer expression is in a proper subrange of the range specified by its type.

A data structure is constructed that represents information about the values of variables and other simple expressions such that if an expression e is mapped to an expression e' in this data structure, then when e is encountered the optimization can rewrite it to e'. Whether or not this happens depends on parameters controlling the complexity of e' vs. e (in order to avoid replacing inexpensive variable references with more expensive expressions). This data structure is propagated forward along the control flow, with merging and fix-point computations occurring at transfers of control and at loops. Substitution of these values is performed when appropriate. In a sense, this transformation is the opposite of common subexpression elimination. Mousetrap performs a pass of limited common subexpression elimination after the last pass through forward analysis in order to repair performance problems that may have been introduced by replicating code unnecessarily.

When a pointer is dereferenced, the assertion that this pointer is not null is established. This assertion is propagated forward to all `case` statements where the tested variable is compared against null; this comparison and the `case` statement itself can then be simplified away.

Destructors ("invalidate" expressions) are then partially inlined to expose more opportunities for optimization. If it is known that an object is non-null, destructors will be transformed as follows: the object is invalidated first and then its memory is freed. The main reason for performing this change is to expose the opportunity to later combine sequences of creation and destruction of memory to avoid unnecessary allocation of new memory. Calling a destructor on a null object has no effect, and the invalidate operation immediately after an initialization can be removed.

Mousetrap also performs several kinds of interprocedural optimizations. Most important is function inlining. If a function is called in sufficiently few places, and is not "too big" (i.e., if the term will not grow by more than a given factor), the inlining transformation replaces each call with an expression that contains a copy of the body of the function. The formal parameters become temporary variables that are assigned the actual parameters. After the inlining transformation has been performed, the term may no longer be in canonical form, so unique renaming and a limited form of expression lifting are again performed.

In another class of transformations, common idioms are recognized and replaced with forms that are more efficient. For example, the optimization recognizes that an SDL String (a list data structure) is being traversed sequentially in a loop. It performs strength reduction, introducing an iterator variable. Because SDL strings are typically implemented as linked lists, this optimization turns a quadratic time algorithm into a linear time algorithm.

Another very important optimization transforms function calls in which some arguments have expensive copy operations into calls where the arguments have only shallow copies. This can be done by introducing pointers or by changing the types of arguments to "demoted" types (for which copy operation do not follow certain internal pointers, but instead continue to refer to the data structure being copied). This optimization is safe only if the source data structure and the copied data structure are not modified during the function call, and if no references to the copied data structure or its components can escape. Alias analysis is used to verify the first of these preconditions, and patterns on the function body are matched. Escape is ruled out by determining that the formal parameter is never used in a referenced L-value in the body of the function.

A similar transformation converts call-by-value on "expensive" types—that is, large structure, union, or array types—into call-by-reference to these types. Side-effect information is used to confirm that this transformation is safe. Linked-list types could be passed by reference as well, but passing by demotion proves to provide better performance. The reason is that variables that have had their addresses taken are less amenable to other optimizations and complicate alias analysis. Structure parameters can exploit a scalarization transformation in which the fields of the structure are passed individually and eliminated if they are always passed constant or globally accessible values, or if they are not used in the called function.

One can also reduce the cost of passing data structures through "structure explosion". In this optimization, variables and actual parameters that are structures and that are not aliased are converted to a set of variables, one for each field of the structure. Assignments to the variable are changed to assignments to its fields, assignments from the variable are changed to structure constructor expressions, and formal parameters are replaced by a set of formal parameters for their fields.

Another means of reducing the cost of manipulating large data structures is to reduce these data structures in size. For example, a transformation looks for array variables that are indexed by values that are in a proper subrange of the declared range of indexes of the array. These arrays are replaced by either smaller arrays of the same dimensionality or arrays of reduced dimensionality (if one or more indexes are always constant).

5.8 Implementation of Core Processes

Instances of a UML active class or an SDL process map directly to instances of an associated Core process type. The efficient and practical implementation of Core process types requires knowledge of the target platform, including the relative costs of threading and synchronization. If a platform provides extremely lightweight threading and synchronization, it may be appropriate to assign each instance of a Core process type to a dedicated thread. In this case, scheduling of process instances reduces to the platform scheduling of threads, and the state of a process instance can be treated as thread-specific data. However, experience with several popular operating systems and with design models with thousands of process instances has shown that the cost of task/thread synchronization eliminates the possibility of one-to-one mapping of threads to process instances. Significant success has been achieved using the simple approach of executing all instances of a Core process type within a single thread.

It is very important to identify the relevant state that must be preserved across the points at which a process instance may block. The naive approach would simply save the values of all process variables that are in-scope at the

points at which a process instance might block. With realistic designs, however, the memory required to implement this strategy can easily exceed hundreds of kilobytes for each process type and overwhelm the capacity of the underlying platform when these storage requirements are multiplied by the thousands of processes that may be active simultaneously. It is therefore essential to minimize the preserved state and to retain the values of only those process variables that are actually "live" at the points at which a process instance might block. Here, "live" means variables whose values may be consumed when a process instance resumes execution after potentially blocking.

Relevant process state is identified through liveness analysis, which utilizes the alias and effects analysis based on sets of abstract locations. The abstract locations that are live prior to the execution of a statement are either the abstract locations used by the statement or the abstract locations used by a successor statement that are not defined (assigned-to) by the statement. As with forward analysis, merging and fixed-point computation must be executed when considering liveness across conditional and loop statements.

5.9 Translation to C

Once optimizations are completed, Mousetrap begins the process of transforming the internal representation to C. One of the products of this transformation stage is a set of header files containing declarations of the types, functions, and global variables of the transformed code. One file contains only the externally visible entities so that this file can be included by anything that needs to invoke functions visible in the external interface.

Integer subrange types are translated to the lowest enclosing C type in the integer type hierarchy of C. This means, for example, that the type `int[0,100000]` is translated to a signed 32-bit integer type because signed types upgrade to unsigned types in C. Appropriate upward casts are added when implementing arithmetic operators. Arbitrary fixed precision integer arithmetic is provided as well, but is much more costly.

At times, it is known that certain branches in the code are unlikely to occur. For example, code can be generated that tests each heap allocation to determine if the request succeeds and jumps to an error handling routine if it failed. This presumably will occur rarely. In this situation, C macros are introduced that inform the C compiler that the relevant `if` statements usually branch in the common direction. Branch prediction is important for generating efficient code on some modern processor architectures.

One peculiarity of C is the ambiguity of nested `if` statements. Abstract syntax trees are syntactically unambiguous, but distinct terms sometimes have the same printed representation. Such terms are recognized by a set of cleanup transformations and are rewritten to forms that print unambiguously in C.

A final transformation pass is performed on the C code to clean up or reduce the size of the generated code. These rules are mostly cosmetic. Examples of cleanup transformations include conversion of `goto` statements into `break` statements or `continue` statements, elimination of unneeded casts, replacement of increment and decrement operations with `++` and `--`, and replacement of `goto`/label loops with `for` loops.

6. Conclusion

In this paper, we have described developing implementations by deriving them through the sequential application of correctness-preserving transformations from high-level models. This technique has been successfully leveraged in the Mousetrap program transformation system to develop application code for Motorola network products.

In contrast to compilers or code translators, Mousetrap performs the transformation from model to implementation in a number of discrete steps that can be adjusted to the application domain and expert user experience. Consequently, program transformation systems such as Mousetrap have been able to consistently meet or beat hand-written code in terms of code quality and performance even in application domains where other ACG approaches have failed: The resultant object size of the generated code for infrastructure network elements has been reduced by as much as 30% when compared with hand-written code; for subscriber devices generated object size is within the same range as hand-written code. Execution speed has met or exceeded performance targets for each developed code generator. When additional performance improvements are needed, transformation rules are easily added to the transformation system to locate and optimize domain-dependent patterns.

In our experience, code generation based on program transformation has reached a level of maturity that allows it to be deployed for the development of performance-critical applications for real-time embedded systems.

7. References

[1] S. Kent, Model Driven Engineering, *Proceedings of Integrated Formal Methods: Third International Conference 2002*, Lecture Notes in Computer Science, 2335, pages 286–298, Springer-Verlag, 2003.

[2] Object Management Group, *Unified Modeling Language (UML)*, Superstructure,Version 2.1.1, 2007.

[3] International Telecommunications Union, *Specification and Description Language*, ITU-T Rec. Z.100, 2000.

[4] T. Weigert and P. Dietz, Automated Generation of Marshalling Code from High-Level Specifications, in System Design, Lecture Notes in Computer Science, pages 374–386, Springer Verlag, 2003.

[5] T. Weigert and F. Weil, Practical Experiences in Using Model-driven Engineering to Develop Trustworthy Computing Systems, in IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, pages 208–217, Taichung, Jun 2006.

[6] J. Boyle, T. Harmer, T. Weigert, and F. Weil, Knowledge-Based Derivation of Programs from Specifications, in N. Bourbakis, *Artificial Intelligence And Automation*, Advanced Series on Artificial Intelligence, pages 315–347, World Scientific Publishers, Singapore, 1996.

[7] T. Weigert, Lessons Learned from Deploying Code Generation in Industrial Projects, *Proceedings of the International Workshop on Software Transformation Systems*, at International Conference on Software Engineering, Los Angeles, 1999.