# Evolution From Debugging to Verification

Frank Rimlinger
National Security Agency
frankrimlinger@mac.com

## ABSTRACT

Can a tool be built that folds code verification directly into the debugging cycle with minimal impact on the code development timeline? This paper describes a precursor tool based on the ACL2 theorem prover, together with suggestions for future work.

## Categories and Subject Descriptors

I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*

## General Terms

symbolic simulation, software verification, automated theorem proving

## 1. INTRODUCTION

Design and development of software has evolved into a rigorous subject worthy of academic instruction. But actually debugging code, making it work, is still an art. Natural selection has produced the programmer artist, whose stock in trade is analytical ability endowed at birth as refined by a personal journey through the dark side of technology. There is a great need to turn the art of debugging into science.

Since a debugger presents a static view of a live execution, that view obviously varies with program input. To migrate the debugging process to verification, it is therefore necessary to debug all the input states. Symbolic execution is a natural means to achieve this end. The literature on this subject is vast, see for example [1] and more recently [2]. This paper uses a language neutral mathematical technique to achieve scalability [8]. A computer program is represented mathematically as a functor from a path category into a category of *predicate transformers*, which intuitively capture the idea of weakest precondition. In this setting, symbolic simulation corresponds to composition of predicate transformers. By manipulating the path category, functorial transformation is used to represent natural code subdivision preserving order of execution and functionality. As subdivided pieces of code are analyzed, the results may be composed to build up a larger picture.

The implemented subdivision process is fully automated and handles submitted code in linear time and space. Therefore, the bulk of the tool described in this paper is focussed on the semi-automated analysis of code fragments. The computer language chosen for analysis is Java, modeled at the byte code level. The tool produces a correct specification of source code level Java as written, modulo proof obligations which are discharged by the ACL2 theorem prover [4]. The crux of the matter is the amount and nature of the interaction required by the user to produce the specification and proof artifacts. The goal is to produce a tool that could be integrated into the code debugging cycle and operated by the programmer, enabling the detection of errors at a time when they are relatively easy to fix. The current tool, in its fourth generation, has not yet achieved this goal. But with each generation, the level of automation increases, and new tasks are identified as candidates for automation. Based on seven years of development resulting in the current tool, the author believes that a sustained evolutionary process will eventually produce a practical interactive tool for code verification.

Section §2 describes the code subdivision algorithm which produces artifacts that drive the symbolic execution mechanism. The basic operation of the tool required to produce the correct specification is briefly summarized in §3, and issues of design drift and compositionality of reasoning are also addressed. Use cases for the existing techniques of the current tool are described in §4. Applications of the existing tool to actual code are described in section §5. Finally, the limitations of the current tool and future work are discussed in section §6.

## 2. CODE SUBDIVISION

The first step in a formal analysis of code is a model of the state of the computer. As the state model for the Java Virtual Machine (JVM) is exceptionally clean, the current tool targets source code written in java and compiled to java byte code. Following [6], the state space of the computer is modeled with just five variables representing the heap, the next free heap address, the stack of frames, the static area, and the state prior to execution of the last instruction.

Given the state model, each byte code instruction is then

a five-valued function of state describing the effect of the operation on each state variable. The modeling approach is similar to that of [7] but was developed independently. For example, an instruction which pops a value and stores it in the heap is represented by a function which emits complex expressions for the heap and stack variables. The first step in creating such an expression is to introduce uninterpreted functions that model the atomic behavior of the JVM, such as accessing the top frame of the stack, accessing the operand stack of a frame, popping a value off an operand stack, and so on. Relationships between such uninterpreted functions may then be introduced as rewrite, or pattern matching, rules. For example, pushing a frame onto a stack, and then popping off this same frame yields the original stack. The cumulative effect of several byte code instructions may then be computed by simplifying the composition of their functional representations using these rewrite rules. This rewriting process therefore accomplishes symbolic execution.

It should be pointed out that this detailed model exists only for the purpose of accomplishing exact symbolic simulation. Various change of variable techniques are applied to the raw results of the simulation prior to visualization and analysis. In particular, proof artifacts generated for ACL2 are, to the extent practical, expressed in terms higher level variables representing source code level values. The heap is presented to ACL2 as an associative array. Although some of the structure of the stack is exposed to ACL2, it can be usually be ignored, the exception occurring during modeling of mutually corecursive methods.

Rather than work in the category of state transitions and composition, it is technically advantageous to work in the dual category of predicate transformers. In this category, described in [8], state transitions, branch conditions, and loops can all be described as predicate transformers. To this end, let $S$ be the state space, and let $I : S \to S$ model a byte code instruction as a map from the state space to itself. Define a *predicate* to be a boolean valued function of state. Let $\mathcal{P}$ be the set of predicates. The the predicate transformer $I^* : \mathcal{P} \to \mathcal{P}$ is defined by the formula $I^*(p) = pI$. In other words, the dual of an instruction takes a predicate evaluated after execution of the instruction to its weakest precondition prior to execution. Give a predicate $b$ modeling a branch condition, the corresponding predicate transformer is defined as $b^*(p) = b$. Finally, let $B$ be a state transition modeling the body of a loop, and let $b$ be the predicate accepting states which enter the loop. Accordingly, a state $s$ will loop $n$ times if $b$ accepts $B^i(s)$ for all $0 <= i < n$ and $b$ rejects $B^n(s)$. The corresponding predicate transformer $\mathcal{L}$ for the loop is defined by the formula

$$\begin{aligned}\mathcal{L}(p)(s) &= pL^n(s) \text{ if state } s \text{ loops } n \text{ times, or} \\ &= \textbf{false} \text{ , if state } s \text{ hangs the loop.}\end{aligned}$$

Thus, the predicate transformer for a loop rejects the states which hang the loop, and otherwise is the weakest precondition.

From this point of view, symbolic execution corresponds to composing predicate transformers along a corresponding execution path. More precisely, given a control flow diagram of state transitions and branch conditions, replace each state transition $I$ with $I^*$, and each branch condition $b$ with $b^*$**and** $\mathrm{id}^*$, where $\mathrm{id} : S \to S$ is the identity state transition. Then composing predicate transformers along any execution path yields a predicate transformer of the form $c^*$**and**$J^*$ such that $c$ accepts all input states which will flow along the path and $J$ represents the state transition along this path. This technique is only effective for code represented by *acyclic* control flow, that is, code which contains no loops. In [8] a sound theory is presented for naturally decomposing control flow into loops with acyclic loop bodies which may refer to other loops in a well-founded, hierarchical manner. This approach was motivated by the work of Legato [5]. This decomposition enables the tool to transform loops in code to recursive functions whose bodies have been simplified via symbolic execution.

The basic idea behind the loop decomposition is recursive *blowup* of *loop clusters*. Starting with an arbitrary control flow diagram, consider the quotient graph obtained by identifying instructions contained within a directed cycle and passing to the transitive closure. The preimage of a nontrivial equivalence class by this relation is a loop cluster. Choose such a cluster, if any, and blow it up by identifying an entry point and adding two copies of it, called $\alpha$ and $\omega$. Redirect all control flow into the entry point to $\omega$, and all control flow leaving the entry point to $\alpha$. Discard the entry point, and repeat this process until the graph is acyclic. Modulo some book-keeping to keep track of control flow and encapsulate loop bodies, the result is the desired hierarchical decomposition.

Observe the word *encapsulate* as used in this paper does not refer to the ACL2 encapsulate command. Rather, to encapsulate a potentially complicated expression $X$ simply means to replace $X$ with a symbol $s$. During symbolic execution, the rewriter may, at its discretion, replace $s$ with $X$. Generally speaking, such replacement occurs if the rewriter wishes to "execute" the code represented by $X$, and otherwise the replacement does not occur. Instead, $s$ simply disappears when the input states which flow into $X$ are excluded from consideration.

Figures 1 and 2 depict an iteration of the blowup process. Once the cluster of loops at $E_1$ has blown up, the single loop at $E_2$ is revealed. The body of the outer loop is represented by the paths from $\alpha_1$ to $\omega_1$. As shown in figure 2, some of these paths still involve edges of the inner loop. However, upon iteration of the blow up algorithm, $E_2$ becomes $\alpha_2$ and $\omega_2$. The body of the inner loop is then defined as the set of paths from $\alpha_2$ to $\omega_2$. As these paths do not meet any other loops, the algorithm grounds out. The paths of the inner loop shrink back to a single point encapsulating the functionality of the inner loop. In this new configuration, the paths of the outer loop no longer meet any loop, just the point $E_2$ endowed with its loop transformer, so once again the algorithm grounds out, and the paths of the outer loop shrink back to a point. Overall control flow is now acyclic, as is the control flow for the body of each synthesized loop. To prevent exponential growth of expressions during symbolic execution rewriting, further encapsulation is performed. In particular, the forward control flow at each branch point in code is encapsulated, so that the rewriter has the opportunity to pause at each decision point, enabling analyst interaction. This interactive capability is described in more

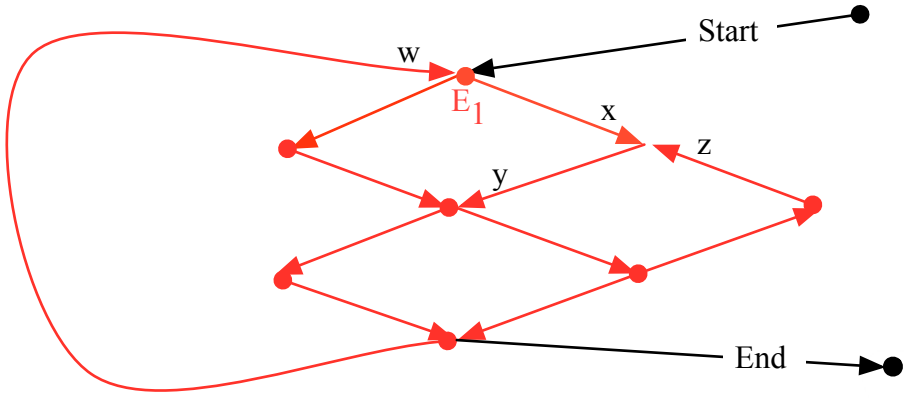


**Figure 1: Loop cluster $E_1$ (in red)**

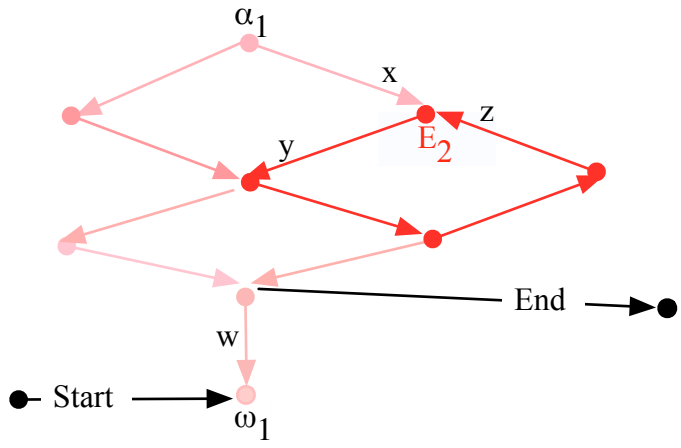


**Figure 2: Blowup of $E_1$ (light and dark red), Loop $E_2$ (dark red)**

detail in §4.

# 3. SUMMARY OF TOOL OPERATION

At start-up, the analyst identifies the source code and corresponding compiled byte code to serve as input to the tool. The tool then automatically applies the code subdivision algorithm to produced the encapsulated functions required for symbolic execution. The object of the analyst is to produce a correct specification of the implemented code. The process is applied on a per *module* basis, where module in this sense refers either to the java concept of a method or a loop body synthesized by the procedure of §2.

The basic iteration of the tool process is referred to as *module definition*. The process produces a persistent set of rules which describe module outcomes together with required input state constraints. In addition, any conjectures required for soundness of the description are present, and well as any change of variables formulas required to translated between JVM level state variables and source code level concepts. With the exception of the change of variable formulas, all of this information translates directly into ACL2, where it is used for the purpose of proving the conjectures.

The module definition process also produces an English language transcription of the outcomes, input constraints, and conjectures. This transcription is not precise enough for formal analysis. Rather, its value lies in the fact that it is immediately intelligible to a tool user with content knowledge of the source code but no special training in formal methods. Experience has demonstrated that many bugs are easily revealed just by reading this transcription to see if it makes sense.

The tool is used iteratively in a bottom up manner to produce module definitions for the entire code body. Initially, module definitions for java native methods are introduced manually into the rulebase. The analyst also has the option of introducing stub modules, which axiomatize the module behavior. The analyst then identifies a module which does depend on a currently undefined module as a candidate for module definition. Accordingly, once this candidate is defined, modules dependent on this candidate may themselves become candidates for definition, and the process repeats until all the modules within the code body are defined.

It should be pointed out that methods which are members of mutually corecursive systems are not defined in isolation according to the above process. Rather, a system of such methods will be preprocessed by the algorithm of §2 into a hierarchical set of loop modules, to be processed by the bottom up procedure summarized above.

A natural consequence of this organization of workflow is a solution for design drift and the compositionality of reasoning problem. Design drift occurs as the details once fresh in the mind begin to fade and be replaced by simpler, potentially problematic constructs. However, whether or not the programmer begins to forget, the computer does not. The module definition of a method is the *only* information about that method that is available to a caller. If a caller cannot automatically relieve the input constraints of a called module, then the anticipated substitution of the module invocation with its instantiated outcome does not occur, and the situation is consequently brought to the attention of the analyst. If the caller input space is suitably constrained so that the constraints of the called method can be relieved, then the substituted functionality for the called method is precisely derived from the module definition in the rulebase. If the programmers understanding has drifted, then the error is potentially revealed during consequent symbolic execution.

Compositionality of reasoning refers to the ability to stitch together separate analytical efforts on subsets of a problem to resolve a larger issue. Not only is the module definition process robust across problem subdivision, but also across different analysts working at different times and places. All that is required is a shared database where the module definitions are logged, and a coordination of effort as required by the bottom up nature of the process.

# 4. TOOL USE CASES

At startup, the tool reads in the rulebase and source code designated by the user. The code is automatically subdivided into predicate transformers according to the method of §2. This boot up process typically takes less than a minute. Operation of the tool proceeds via interaction with the user. This section describes in roughly chronological order a set of use cases for tool operation which result in the generation of a English language specification for a module and the associated proof artifacts for ACL2.

## 4.1 Select module for definition

The predicate transformers representing the functionality of the modules processed at boot up are presented to the user in a hierarchical display reflecting the package structure of java code. Embedded within the package hierarchy is the hierarchy of module dependence generated by the tool. Only modules not yet defined in the rulebase, in the sense of section §3, are exhibited in this manner. Accordingly, the user navigates to a leaf node in this hierarchy to select the next candidate for module definition. This causes a window to appear which displays the current status of the definition. From this display the user can access the current predicate transformer associated with the module, as well as hypotheses, outcomes, and conjectures associated with any previous cases. Once all desired cases for a module have been defined, then the module definition is closed and will no longer appear in the initial display.

## 4.2 Inspect an expression

The user guides the tool in the process of simplifying the current module expression. In order to do this, the expression must be presented to the user in a comprehensible format. The low level language expressing a predicate transformer as a function of the JVM state variables is not suitable for this purpose. The high level language involving change of variables stored persistently in the rulebase and exposed to ACL2 is not yet available, as the change of variables transformation itself is not yet defined. What can be displayed is an intermediate language based on debugging information that is typically stored in the compiled java .class files. In particular, the mapping of local variable offsets to source code level names is leveraged to effect a transformation to

source code level constructs. Inspection of this expression allows the user to track the path through the code that has been accomplished so far by the symbolic execution. All expressions are displayed hierarchically and are fully navigable. This feature allows for easy access to subexpressions, which is required by various use cases below.

## 4.3 Step the symbolic execution

The granularity of the code subdivision process is advertised in §2 as being at the level of branch points in code. Ideally, this would imply that every time the user steps the rewriter, the current module expression advances to the next branch point. However, for a variety of technical reasons, some stuttering does occur. Once a branch point, module invocation point, or execution terminus is reached, the command to rewrite will no longer alter the current module expression, and the user will decide on a course of action as detailed in the following use cases. An exception to this rule is the situation of automatically generated hypothesis, discussed in the next use case.

## 4.4 Choose a branch

Logically, choosing a particular branch in the code means asserting a hypothesis about the module input state. This hypothesis is recorded in the definition window for the module, and ultimately becomes an input constraint in the rulebase for the case under construction. The hypothesis also becomes available immediately to the rewriter, so that subsequent rewriting will cause the symbolic execution to advance. In addition, the English language translation of the hypothesis is displayed in a running transcript, so that the user can keep in sync with what has been assumed. In practice, the display of the English transcription seems to have a powerful psychological effect on the user, even though it has no logical significance. It is empirically evident that seeing the same information displayed in different ways within the same context enhances comprehension.

There are many very routine hypotheses that the tool will assume on its own initiative. For example, there are a variety of reasons why an invocation exception might be thrown. The logic for such an exception is built into the state transformations of the various invocation instructions at the JVM level. From a purely formal standpoint, there is no difference between a constraint that assumes such an invocation exception does not occur and a constraint involving the logic of the program evident at the source code level. However, it is essential not to involve the user with routine, boring activity which causes attention to wander. Therefore, the rewriter will automatically cause such routine conditions to be assumed. The English transcriptions of these assumptions appear in italic so that the user can more easily focus on the significant constraints.

It is well to point out that the term *rewriter* as used in this paper goes well beyond the basic functionality of altering expressions via a pattern matching mechanism. The rewriter actually functions as a mini operating system, capable of acting upon events which arise during the pattern matching process and scheduling appropriate tasks to service those events. By this means, the technical apparatus for achieving automation of formally identifiable tasks is built into the tool mechanism.

## 4.5 Create a user definition

There is an ever present danger of expression complexity explosion during the process of rewriting. Most of the time, the rewriter itself can take effective means to mitigate this problem. However, there are times when the user may wish to intervene by encapsulating a subexpression as a *user definition*. There are four categories of required rules for a user definition: declaration, evaluation, inversion, and translation. The user must supply rules in the rulebase to satisfy each of these categories. The declaration rule provides a type and name for each of the arguments, and a type and name for the definition itself. The evaluation rules determine how the definition will expand when evaluated. The inversion rules determine how an inverted expression or subexpression will transform itself into an instance of the definition. Finally, the translation rules determine how an instantiated definition expression will translate into English, assuming inductively that each of the arguments already possesses such a translation.

Needless to say, there is ample opportunity for error during the construction of a user definition. Experience has shown that the process of translating user definitions to ACL2 will expose such errors. This appears to be one area where non-sophisticated ACL2 users can use the tool with relative ease and effectiveness.

## 4.6 Invert or evaluate an expression

User definitions have no effect on rewriting unless the inversion or evaluation mode is selected. Most of the time, it is advantageous to rewrite with inversion, so that complex expression are immediately bundled up into instantiated user definitions. Therefore, inversion is the default mode for symbolic simulation. However, situations do arise where other courses of action are more appropriate. Because the expressions are fully navigable, the user can mix and match, choosing to invert or evaluate only those subexpressions that are under investigation.

## 4.7 Resolve an instantiation

The rewriter detects invocations of defined modules and fires up a handler to service the invocation. The handler provides visual feedback by inserting the name of the called module within an appropriate panel of the module definition window. The handler then inspects the cases for the defined module within the rulebase. The goal is to find a case whose hypotheses are satisfied by the state passed to the invocation. To this end, the hypotheses are first converted back to JVM state expressions using the change of variables rules for the defined module. The hypotheses are then composed with the input state, and rewritten in both inversion and evaluation mode. If each hypothesis for a case succeeds by some means in rewriting to **true**, then the handler substitutes the instantiated outcome for the module invocation within the current model expression, and rewriting continues. Otherwise, the handler opens up an new instantiation window for the called module.

This is a dramatic event that commands the attention of the user. The user may access the instantiated hypotheses and outcomes via this window, and determine a course of action. The user may attempt to relieve hypotheses by more sophisticated use of the rewrite commands. Alternatively, the user

may decide to *assume* the unrelieved hypotheses for a particular case. The user terminates the instantiated event by closing the instantiation window. If the user closes the instantiation window prior to assuming or otherwise relieving the hypotheses of a case, then an instantiated module definition is substituted for the invocation. Instantiated module definitions are syntactically similar to instantiated user definitions, but they are treated in a special manner by the rewriter because they are of type *state*. The significance of the state type is further discussed in the next use case.

## 4.8 Generate an invariant

Prior to rewriting, the current predicate transformer of a module definition in progress is instantiated with a dummy state. The rewriter treats accessors of dummy state in a special manner, rendering such expressions using available context, as discussed in §4.2. As discussed above, the failure to resolve module instantiation results in module state expressions within the current predicate transformer. As the rewriter detects accessors of such expressions, it renders them using both context and the name of the called module. So, for example, an access at an offset into the local variable array of the top frame of the stack of a module state expressions might render as "x after call to methodName". The appearance of such variables is an indication that the user should create an *invariant*. It may be that the called module does not alter the value of x. The user may introduce an invariant rule into the definition of the called method, which rewrites "x after the call" to "x before the call", or to whatever value is suitable under the circumstances. The tool then assumes the truth of such invariants during rewriting. However, the possibility of inconsistency exists until the obligation to prove the invariant is relieved using ACL2. The benefit of uncoupling the proof activity from the specification activity is a more cost effective division of labor and increased throughput.

Invariants typically are straightforward substitutions, but they may be quite complex, depending on the functionality of the intervening module. These more complex invariants are sometimes referred to as *conjectures*, but their logical status is the same.

## 4.9 Simplify the heap

The temporal logic involved in garbage collection is a subject worthy of study. The current tool recognizes two different kinds of garbage, heap access garbage and output state garbage. For heap access, all heap entries with reference provably unequal to the accessor reference may be eliminated from the heap access expression. This is a particularly valuable strategy for demonstrating syntactic equality of unresolvable heap access expressions. When a method returns, all items in the heap that were created during execution of the called method are stale, with the exception of any items that may be traced from a reference returned by the method. The tool uses this principle to garbage collect the output heap generated by the module definition process for the special case the module is a method. For both kinds of garbage it is essential to bind each reference with a temporal tag indicating if it is *in the past*, that is, referring to an object created before the call to the current module, or *in the present*, that is, referring to an object created during execution of the current module, or null, static, or unknown. Heap access garbage is collected automatically by the rewriter. Output state garbage is collected by user command just prior to closing a method case or a definition.

## 4.10 Close a case

Once the current module predicate transformer has rewritten to a state containing no uncomposed subexpressions, the entire expression is *a fortiori* a function of dummy state. As such, it may be stored in the rulebase as *case*, together with its associated hypotheses, invariants, and conjectures. The user initiates the close case command.

## 4.11 Close a definition

The user issues the close definition command when closing the final case of a module definition. In addition to storing the apparatus of the final case, the tool automatically performs the change of variables analysis. This involves a survey of all persistently stored expressions associated with the module definition, to determine all state accessors, and all stack and heap expressions. Each such accessor is assigned, if possible, a source code level name. The type of the accessor is also determined. Each such accessor function is now regarded as a change of variables formula for an associated parameter. An analysis is performed to determine functional dependence of parameters. This process grounds out in parameters that have change of variable formulas that are functions of pure state. Other parameters might have less dependent parameters as part of their change of variables formula. For example, the heap parameter might be a function of local variable parameters it contains as heap references. The point of preserving relationships between parameters is to allow this information to be translated into ACL2, to the extent it is necessary to do so to perform the required analysis within the theorem prover.

Once the set of parameters has been established for a module definition, the all the persistent expressions associated with the module definition, are transformed to their parameter form, with the obvious exception of the change of variable formulas themselves. The module definition at this point is marked as closed, and no more cases can be added to the definition.

## 4.12 Examine English language specification

The English language specification has been referred to in various places throughout this paper, but it still deserves a use case of its own. The author originally added this functionality to the tool as a marketing ploy, but it was immediately apparent that translation to English is a surprisingly effective means of communicating meaning. In time, this translation may become natural enough to replace the javadoc that is now generated by totally unscientific means to describe method functionality.

## 4.13 Translate module definitions to ACL2

The current method of translating to ACL2 is simply to paste expressions into the system and then try to sort out all the problems. This is an area that hopefully will be getting a lot of attention in the near future. Suffice to say that after several iterations of attempting to generate data for ACL2, the author is now confident that the correct information is now stored in the rulebase in persistent form, and that

the remaining problems are more technical in nature than conceptual.

## 5. PROOF OF CONCEPT

The ability of the tool to communicate effectively with ACL2 has been a constant theme throughout the development of the tool. The change of variable techniques to translate between JVM level and source code level language were first introduced to facilitate this communication. Earlier versions of the tool did not integrate change of variables into the module definitions stored persistently in the rulebase. Instead, to accomplish the translation to ACL2, an additional persistent layer of language was stored in the rule base, and a parser written in lisp was developed by Robert Krug to translate this information into ACL2. Although this method was successful, it was also clumsy and fragile. The current tool integrates change of variables directly into module definition, so that translation ACL2 should be much more straightforward. As of this writing, testing of this new approach is just getting underway.

Using the earlier approach, Krug applied the tool to a set of java programs involving a simple loop to sum a sequence of integers, a loop to zero the array of a heap, and a triply nested summation loop. Conjectures generated by the tool about these programs, together with the module definitions, were translated into ACL2 using the parser. The conjectures were then proven within ACL2. These tests demonstrated the basic ability to translate simple loops to ACL2, but did not address the problem of scalability. A very significant milestone for the tool was the proof of correctness of the code for java.lang.System.arraycopy(). This method contains a loop to transfer data from a specified range within one array to a specified range within another array. The code potentially can generate a variety of error conditions involving range checking and data compatibility. The resulting module definition generated by the tool for arraycopy() was therefore significantly more complex than the earlier test cases. The tool generated an appropriate conjecture for the correctness of the arraycopy() routine. The entire situation was translated to ACL2 and the conjecture was proved therein.

In addition to the code base used to drive tool development, the tool has been applied to about 500 lines of high quality code that had already undergone a rigorous review involving code walkthroughs and testing. (A line of code is defined to be a source code line containing at least one semi-colon.) No errors were detected in code functionality. However, 17 errors were detected within the supplied java doc describing method functionality. Five instances of unintended, uncaught behavior were detected. These were not considered outright implementation errors because they involved method input thought to be "illegal".

## 6. FUTURE WORK

The direction of further work on the tool is towards an extension of the normal debugging process which a programmer may exploit to compare the implementation of code as it is written with any pre-existing design concept. In order to lure a programmer into using such a tool, it is essential to make it appear as familiar as possible, at least on a superficial level. To this end, the popular open source framework provided by Eclipse [3] seems like an ideal delivery platform. The current rendering of formulas takes many hours of self-training to easily comprehend, and so must be completely reworked for display within an Eclipse perspective. Ideally, formulas should have a top level display which mimics the familiar variable/value debugger format. The lower level description in terms of state transition and branch condition can always be revealed as necessary using level-of-detail technology.

Another long range goal is the building of a large set of significant test cases for ACL2 translation and relief of conjectures within ACL2. Now that the framework for this translation has settled down, it should be possible to proceed with specification and verification of the java library code, which contains many excellent examples of complex loop behavior. Needless to say, specification of the critical core code within the java libraries is itself an essential prerequisite for deploying the tool as part of a development environment.

## 7. REFERENCES


[1] Robert S. Boyer, Bernard Elspas, Karl N. Levitt, *SELECT-a formal system for testing and debugging programs by symbolic execution*, ACM SIGPLAN Notices, v.10 n.6, p.234-245, June 1975

[2] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, Mauro Pezze, *Using Symbolic Execution for Verifying Safety-Critical Systems* Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pages 142-141, 2001

[3] *Eclipse Platform Plug-in Developer Guide*, Eclipse Foundation `http://www.eclipse.org/documentation` 2007

[4] M. Kaufmann, P. Manoios, J. S. Moore, *Computer-Aided Reasoning, An Approach*, Advances In Formal Methods Serieis, Kluwer Academic Publishers, 2000

[5] W. Legato *Generic Theories as Proof Strategies: A Case Study for Weakest Precondition Style Proofs*, Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004), available on-line at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/legato/Generic-Theories-as-Proof-Strategies-Report.pdf`

[6] T. Lindholm and F. Yellin, *The Java™ Virtual Machine Specification*, Second Edition Sun MicroSystems, Inc, 1999

[7] J. Moore, R. Krug, H. Liu, G. Porter, *Formal Models of Java at the JVM level– A Survey from the ACL2 Perspective*, Workshop on Formal Techniques for Java Programs, in association with ECOOP 2001, June, 2001 (long version)

[8] F. Rimlinger, *A Theory of Assurance*, submitted to Logical Methods in Computer Science, available on-line at `http://idisk.mac.com/frankrimlinger-Public`