On Formal Definition and Analysis of Formal Verification Processes

Leon J. Osterweil

Lab. For Advanced Software Engineering Research
(laser.cs.umas.edu)
School of Computer Science
University of Massachusetts
Amherst, MA 01003
ljo@cs.umass.edu

Abstract. This paper suggests that there is considerable value in creating precise and formally-defined specifications of processes for carrying out formal verification, and in then subjecting those processes to rigorous analysis, and using the processes to guide the actual performance of formal verification. The paper suggests that some of the value could derive from widening the community of verifiers by having a process definition guide the performance of formal verification by newcomers or those who may be overawed by the complexities of formal verification. The paper also suggests that formally-defined process definitions can be of value both to novices and more experienced verifiers by serving as subjects of both dynamic and static analyses, with these analyses helping to build the confidence of various stakeholder groups (including the verifiers themselves) in the correct performance of the process and hence the correctness of the verification results. This paper is a status report on early work aimed at developing such processes, and demonstrating the feasibility and value of such analyses. The paper provides an example of a formally-defined verification process and suggests some kinds of dynamic and static analyses of the process. The process incorporates specification of both the nominal, ideal process as well as how the process must be iterated in response to such verification contingencies as incorrect assertions, incorrectly stated lemmas, and failed proofs of lemmas. In demonstrating how static analyses of this process can demonstrate that it assures certain kinds of desirable behaviors, the paper demonstrates an approach to providing effective verification guidance that assures sound verification results.

1 Introduction

Society is becoming ever more dependent upon systems that rely importantly upon the reliably correct functioning of software. Air travel is now heavily dependent upon software that is pervasive both in the cockpit and on the ground. Automobiles are similarly increasingly dependent upon software. Health care devices and systems also increasingly employ software in critical roles. Because of this it is correspondingly important that the software in these systems performs correctly, across the

S. Iida, J. Meseguer, and K. Ogata (Eds.): Futatsugi Festschrift, LNCS 8373, pp. 35–52, 2014.

increasingly broad spectrum of situations in which it is relied upon. There are many approaches to assuring the correct functioning of software, including dynamic testing and various forms of static analysis. But perhaps the strongest assurances of correct functioning of software are provided by formal verification. This paper describes an approach that is currently being developed for making the performance of formal verification accessible to more practitioners by providing explicit proactive process guidance to this performance.

In addition, we note that while it is essential that critical software reliably function correctly in all circumstances and situations, it is also most important that those who rely upon such correct functioning have a sound basis for believing that that is the case. In short, it is not enough for software to perform correctly, but it is also important that all of the software's stakeholders have access to satisfactory evidence of this correct performance. This paper also suggests that the explicit process guidance provided by the approaches presented in this paper can also be the basis for providing importance forms of evidence that the formal verification process has been performed correctly. Evidence of correct functioning that is provided by testing and many forms of static analysis can be relatively accessible to broad communities of stakeholders, but it seems important to also consider how it might be possible to provide broadly accessible evidence of correct performance of formal verification. It is the position of this paper that the formal verification community has made good progress towards the goal of understanding how to reason about the correct performance of software (although it would be desirable to make formal verification processes more accessible to more practitioners), but that more progress should be made towards the second goal of being able to provide convincing assurances to stakeholders that a formal verification process has actually been carried out correctly.

Over the past several decades many approaches to reasoning about the correct performance of software have been developed. The most widely recognized and employed approach has been testing, where the program is run using a large number of diverse input datasets. Typically the results can be examined by diverse stakeholder communities to provide these stakeholders assurances that the program behaves correctly. While this approach allows for the in-depth exploration of the behavior of a program under actual runtime conditions, testing results do not extrapolate. Thus, even if a program has been run successfully millions of times on different datasets, there is no assurance that it might not fail on the next test execution. Consequently testing is unable to offer the kinds of ironclad assurances of correct performance that are required by many stakeholders in software for critical applications.

Static analysis approaches such as finite state verification [3] and model checking [1] can offer some kinds of more definitive assurances, however, and thus serve as a useful complement to dynamic testing. The static analysis approach makes it possible to prove that all possible executions of a program must necessarily always satisfy certain classes of properties. Typically these kinds of properties are modeled as sequences of events, often represented by finite state machines. Insofar as these properties are modeled by relatively accessible diagrams such as finite state machines, and anomalous execution sequences can be presented as statement execution traces, these kinds of analyses can provide relatively accessible evidence of correct program performance to relatively broad classes of stakeholders. A principal drawback of this approach, however, is that the need to represent these properties by formalisms such

as finite state machines limits the kinds of properties for which it is useful. In particular these static analysis approaches are generally not useful in supporting demonstrations that the program will always necessarily demonstrate the desired functional behavior.

Especially in view of the foregoing, formal verification [2,4,7] occupies a very important position among the many approaches to reasoning about the behavior of programs. Formal verification can be used to prove that all possible executions of a program must necessarily always deliver specified functional behavior. Doing so, however, requires a considerable amount of effort and resources. The program to be formally verified must be written in a language whose semantics have been defined formally, the specified behavior must be defined in the form of formally specified assertions, and human verifiers are invariably required to create large numbers of proofs, each of which may be quite complex and must be meticulously reasoned. Because the reasoning process is complex and lengthy, errors of many kinds can be committed. The structure of lemmas to be proven may be flawed, the assertions essential to the statements of the lemmas may also be flawed, and the details of the actual proofs may be incorrect. All of these difficulties have in the past served as obstacles to the broader adoption of formal verification by practitioners, and to the accessibility of formal verification results by broader stakeholder communities. Novices, in particular, have all too often been daunted by the complexities of performing formal verification. And the complexity of the work of formal verification experts has at times seemed to be beyond the grasp of some stakeholders, suggesting the desirability of additional forms of visibility into how the formal verification process was carried out. In both cases, the continued evolution of software systems creates additional difficulties. Once modified in any way, a previously-verified program must be reverified. If modifications are minor and quarantined to a small program locality, the reverification of the entire program may not be necessary. But it can be difficult to determine which reverifications are necessary, and which are not. Automated tool support can be quite useful in guiding both novices and experts to the correct determination.

Especially since formal verification is employed most commonly to offer the most solid assurances of correct performance to the most demanding stakeholders, it then seems appropriate that these stakeholders have the most definitive assurances that the verification results are trustworthy. In large-scale industrial contexts, these assurances are often obtainable using verification assistants and checkers such as Isabelle [12]. But even in these contexts, the need to reverify software as it continues to evolve can lead to uncertainty about just which portions of which versions of a program have been subjected to which verification activities. In other contexts, especially where verification is done informally or by novices, these assurances are harder to obtain, and necessarily less reliable.

A variety of directions have been taken in order to address the many difficulties inherent in formal verifications of programs. Of particular interest in the context of this event, we note that Futatsugi [5, 6] has suggested the value of verifying designs rather than code, inventing the notion of Proof Scores. Another approach has advocated the use of automated proof checkers and proof assistants to support humans in carrying out the formal verification process. The first tool to provide such support, a verification assistant, was developed in 1969 by James King [8]. Many such systems have

been developed subsequently. Currently Isabelle/HOL [12] seems particularly popular and effective in supporting the verification process and the checking of needed proofs. It is important to note, however, that the participation of humans is typically essential, especially at the higher levels of lemma specification, and that considerable amounts of iteration and rework are typically required to successfully complete an entire formal verification. Typically humans must create and place assertions, and guide the proof of many lemmas derived from the assertions and program code. As noted above, iteration is typically necessary, requiring and responding to assertion modifications, lemma regeneration, and proof defects. As also noted above, program evolution necessitates reverification that can be expedited by lemma reuse, but can also lead to configuration management issues leading to mistakenly thinking that an incorrect verification is correct.

One approach to addressing these problems is to formally define formal verification processes that incorporate specifications of these various kinds of iteration, and then to apply the various forms of reasoning just summarized to this process definition. In short, we advocate formally defining realistic iterative formal verification processes, and then applying dynamic testing, finite state verification, and other forms of analysis to such processes in order to generate analysis results that can lead to greater insight into these processes, and increased credence in the results they deliver. The purpose of this paper is to indicate that such formal definitions and analyses of formal verification processes are feasible, and should be increasingly important additions to the formal verification enterprise. To that end, this paper describes early work that is developing formal definitions of iterative formal verification processes such as Floyd's Method. The paper presents one such process definition that has been written in a rigorously-defined process definition language. Because of the language's semantic definition in terms of a rigorous notation (in this case it is Finite State Machines) we are able to demonstrate the feasibility of applying rigorous analysis to these processes, thereby obtaining definitive analytic results.

2 A Process-Centric View of Formal Verification

A formal verification of a program is essentially a proof that all possible executions of the program must necessarily deliver functional results that are consistent with a specification. The program and a precise specification of desired functionality are taken as input to the verification process, and the desired result of the process is a proof of a theorem that the program meets its specification. The process of producing the proof consists of creating and then proving a carefully constructed set of lemmas. We will use Floyd's Method of Inductive Assertions [4] as an example of an approach to formal verification.

2.1 Floyd's Method of Inductive Assertions

Floyd's Method begins by creating a set of assertions, each of which is a statement that characterizes what should be true at a specific point in the execution of a

program. These assertions must then be placed so that every program loop is "broken" or "cut" by an assertion (i.e. every iteration of every loop in the program must necessarily encounter at least one assertion). Initial and Final Assertions are also placed at the beginning and end of the program to capture the desired functional behavior of the program. The placement of these assertions assures that every possible execution of the program is a sequence of loop-free statement execution sequences, each of which is bounded at each end by an assertion. Because any program has a finite number of statements there are only a finite number of places where assertions can be positioned. Thus there are a finite number, N, of assertions placed in any program, and there can therefore be at most N^2 pairs of assertions. Assuming that there are at most C different paths between any pair of adjacent assertions, then any execution of the program can be viewed as a sequence that consists of at most $C*N^2$ different loop-free statement execution sequences that are bounded at each end by an assertion. Assuming that there are in fact L such different loop-free execution sequences (where $L <= C*N^2$), then the essence of Floyd's Method is to prove L lemmas, each of which consists of demonstrating that, assuming the assertion at the start of the loop-free execution sequence is true, the execution of the loop-free execution sequence then assures that the assertion at the end of the loop-free execution sequence must also be true. If all such lemmas can be proven, then by induction, for any possible program execution, the truth of the initial assertion guarantees the truth of the final assertion assuming that the execution of the program reaches the final assertion. The verification of the program then requires a proof that the program must actually terminate.

This elegant approach to reasoning about the functionality of a program provides an excellent intellectual framework for understanding and reasoning about a program. The need to be sure that the final assertion is implied by all of its possible predecessor assertions makes it imperative that all of these previous assertions address the real functional substance of the program (e.g. trivial assertions such as *true = true ^ true* will not help imply the final assertion). Thus there is strong pressure for intermediate assertions that are placed inside of loops to be specifications of the quintessential contribution that each loop iteration makes to the overall work of the program. As such, creating these so-called *loop invariant assertions, or loop breakers,* compels the human verifier to come to grips with the nature of the program being verified, and to develop a deep understanding of the program. This discipline is widely regarded as being of at least as much value and importance as the actual completed proof itself, suggesting that it is particularly important for novice programmers to understand and practice the discipline. One key motivation of the work described here is to suggest an approach to helping novices feel comfortable in carrying out formal verifications of their programs.

2.2 Pragmatic Issues

While the conceptual basis for Floyd's Method is beautiful and elegant, the actual execution of the method is fraught with difficulties and perils. Ultimately the human verifier would like to either prove all of the lemmas, thereby verifying that the program must always produce the correct functional results, or in failing to do so come to

the realization of the existence of a program error that must be fixed. But simply failing to be able to prove all lemmas may result from difficulties in performing the verification, rather than from the presence in the program of an error that must be fixed. Specifically, the following are some of the difficulties that a human verifier may encounter in performing the verification:

- An assertion may be incorrect or inadequate: As noted above, a loop invariant assertion must capture the essence of what each loop is quintessentially all about and what it contributes to the overall functioning of the program. All too often programmers have only a fuzzy grasp of this and may build loops whose actual functioning lacks this sharp focus. In such cases, the verifier will understandably have difficulty in enunciating simple and elegant loop-breaker assertions. Sometimes this lack of clarity of purpose causes a loop indeed to be programmed incorrectly, but often it simply creates an intellectual challenge that requires the verifier to iterate the specification of the loop invariant or modify the program code, seeking code and assertion that will suffice. Thus, lack of success in proving a needed lemma may well indicate ways in which a loop invariant specification may need improvement or code should be modified. When any code or assertion is changed, then it becomes necessary to repeat the proof of any previously-proven lemma that involved that assertion or code.
- An assertion may be positioned in the wrong place in the program: In a similar way, it may be the case that the verifier understands the essential nature of the performance of a loop, but may position a needed loop invariant in a location where the specification of the invariant it not true under all circumstances. Here too failure to prove one or more lemmas may lead to a clearer understanding of where the assertion needs to be positioned. And here too, any change in the position of an assertion requires that any lemma involving that assertion be reproven.
- The proof of a lemma may be incorrect: The proof of a lemma requires that the contribution of every statement in the loop-free execution sequence be characterized as a specific change that the execution of that statement makes to the overall state of the program's execution. The statement's contribution may be to change the value of one or more program variables, or the concurrency state of the program, for example. The semantics of the language in which the program is written provide a template for determining the contribution of a statement, and the specifics of each statement can then be used to elaborate the template into a precise and detailed specification of the contribution of the statement. Proving a lemma entails composing the contributions of each of the statements in the loop-free execution sequence, and proving that their combined behaviors must always cause the final assertion to be true, given that the initial assertion is true. Because the semantics of a typical programming language are defined using non-trivial mathematics, proofs of lemmas about programs written in such languages require good facility with such mathematics. Even a very minor error in inference can cause a verifier to incorrectly conclude that a correct lemma is incorrect, or that an incorrect lemma is correct. Either of these errors invalidates the entire verification. Because of this,

proof checking tools are typically used to review the proofs of lemmas. If the proof of a lemma is found to be incorrect, the proof must be corrected.

- The lemma may be correct, but too difficult for the verifier to prove: As noted above, a proof typically requires reasoning meticulously about the combined behaviors of all of a sequence of program statements. When the sequence of statements gets long, the combined behaviors of these statements can become quite complex, requiring the verifier to create a long and complex lemma. The very size and complexity of the lemma may make the proof too difficult for the verifier to carry out. In such cases, the verifier may expedite the verification process by creating new assertions, modifying some of the existing assertions, moving the locations of some of the existing assertions, or modifying the underlying program to be verified. In each of these cases, the changes to assertions or to program statements will require reconsideration of the previously developed lemmas and proofs.

The foregoing suggests why many novices may be daunted by the prospect of carrying out the formal verification of a program. But it also suggests the value of proactive process guidance through the different kinds of iterations that may be necessary, and the prospect that such guidance could increase the accessibility and appeal of formal verification to novices. In particular, because the formal verification of a program is likely to require a considerable amount of trial and error, leading to a considerable amount of iteration, guidance through these iterations could be of considerable value. Some iteration might be minor, requiring only a minor modification to a slightly flawed proof of a single lemma. But some iteration, such as the need to modify an assertion that is a part of several different lemmas, may require a very considerable amount of effort. In cases where considerable effort seems required, it is reasonable for the verifier to think carefully about which lemmas need to be reproven, and which need not be reproven. In the case of lemmas to be reproven, it is reasonable for the verifier to seek to reuse as much of the previous proof as possible. All of this reasoning is error-prone and can lead to the incorporation into the final set of lemmas of one or more lemmas whose correctness might be incorrectly assumed, leading to an incorrect verification. Therefore, proactive support for reasoning about he reuse of lemmas would be of great value to the verifier.

Documentation of the reasoning about lemma reuse can also be of considerable value to stakeholders. The possibility that a formal verification of a program is incorrect should be a concern for all of the stakeholders of that program. All of these stakeholders should be insistent upon having access to evidence that the verification has indeed delivered correct results. Among the kinds of stakeholders that should have this concern and should require such evidence are:

- Customers who have paid for the program and should expect that they are receiving what they have paid for.
- Users who need the functionality that has been promised.
- Innocent bystanders (e.g. passengers on a software-guided airplane) whose safety might be jeopardized by software that performs in an incorrect and/or unsafe way.
- Developers whose pride and professional reputations derive from their demonstrated ability to create software that meets the needs of stakeholders.

There are some differences in the ways in which the needs of these different stakeholders should be met. But all are seeking assurances that the process by which the formal verification of the program was carried out is itself correct, and was carried out correctly. Our view is that these assurances can be derived from appropriate examinations of a sufficiently precise and detailed definition of the formal verification process, and of a sufficiently precise and detailed history its execution. Thus, for example, it would be important to be able to show definitive evidence that a proof checker has been run on each of the lemmas. And to show that none of the assertions or code involved in a lemma that is incorporated as part of a final program verification has been modified subsequent to the running of a check of its proof. As noted above, automated tool systems such as Isabelle can help to provide these kinds of evidence. But as a less expensive and more accessible alternative, an appropriately complete and detailed definition of the formal verification process could be used to provide some of these forms of evidence as well. Moreover, a trace of the execution of the process could provide evidence that all proofs were indeed checked, and no changes were made subsequently. Human verifiers' efforts could be augmented by such a process if the process were to specify that the human verifier could not declare the program to be verified until and unless all lemmas were proof-checked. These examples suggest ways in which an appropriate formal verification process definition could be used to provide desired assurances of different kinds to different stakeholder groups.

3 An Example Formalization of a Formal Verification Process

To demonstrate the feasibility of creating a formal definition of a formal verification process, we now use Little-JIL, a semantically well-defined language [13,14], to define precisely and in some detail Floyd's Method of Inductive Assertions. We have chosen to use Little-JIL as the vehicle for the definition of this process for a number of reasons. First, the semantic scope of Little-JIL seems particularly well-matched to the needs of an iteration-intensive formal verification process definition in that Little-JIL provides particularly strong support for such features as abstraction, exception management, rigorously-defined artifact flow, and human choice, all of which seem to be important in formal verification processes. Thus, for example, Petri Nets seem particularly poorly suited to the concise specification of such processes in that they lack strong features for modeling artifacts and their flow, and are particularly clumsy in dealing with exception management. In their lack of hierarchical structure they make it hard to deal with abstraction. Other specification languages have other patterns of weakness in specifying these critical features of iteration-intensive formal verification processes.

3.1 About Little-JIL

We now provide some minimal level of details about the Little-JIL language. More complete details about the language can be found in [13] and will also be introduced in the context of our explanation of the definition of Floyd's Method. Little-JIL is a visual language, depicting processes as hierarchies of steps. But Little-JIL is also

semantically well-defined, with its semantics being based upon finite state machines. The semantics are sufficiently strongly defined to support the execution of processes defined in Little-JIL. A presentation of the finite state machines used to define Little-JIL semantics is well beyond the scope of this paper. But these finite state machine definitions have been used to support various forms of reasoning about processes from many diverse domains that have been defined in Little-JIL. In this section we build upon that process reasoning experience to apply it to reasoning about formal verification processes.

A Little-JIL process definition looks initially somewhat like a task decomposition graph, in which processes are decomposed hierarchically into steps, with the order of execution of child steps being specified by the parent. Steps can be thought of as procedures, especially in that they incorporate specifications of argument flow. When a Little-JIL process definition is executed, its various steps are elaborated at run time into step instances.

Figure 1 is a visualization of a step, where the black bar represents the step. Little-JIL steps are connected to each other with edges that represent both hierarchical decomposition and artifact flow. These edges are shown emanating from the left side of the step bar in Figure 1. The left side of a non-leaf step bar contains an iconic representation of the order in which the step's substeps are to be executed. Little-JIL incorporates four different step execution sequencing specifications: sequential (indicated by a right facing arrow), which specifies that substeps are to executed sequentially from left to right; parallel (indicated by an = sign), which specifies fork-and-join for its substeps; choice (indicated by a circle slashed through the middle), which specifies that only one of the step's substeps is to be executed, with the choice being made by the parent step; and try (indicated by a right facing arrow with an X on its tail), which specifies that the step's substeps are to be executed in left-to-right order until one of them succeeds by failing to throw an exception.

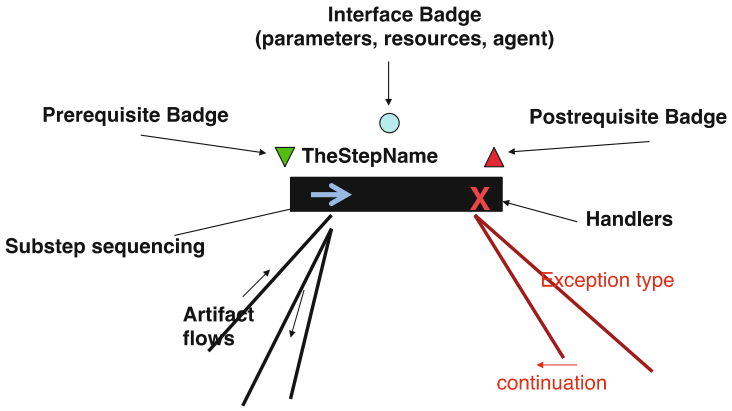

Fig. 1. A visualization of a Little-JIL step showing its key semantic features

Each step definition contains an interface specification, represented in Figure 1 by the small blue ball over the step bar. The interface specification consists of a specification of the step's arguments and its resource requirements. An argument specification incorporates both type information and information about whether the argument is an input, an output, or both. Step invocation parameter-passing semantics are essentially copy-and-restore. A step's resource requirement specification details the types of resources needed in order to perform the task associated with that step. In Little-JIL, moreover, one resource is always designated as the step's agent, namely the resource responsible for the performance of the step. Thus, for example, in Floyd's Method, the agent for a step such as the specification of a loop invariant would be a human, but the agent for a step that checks the details of a proof would probably be an automated proof-checker.

Exception handling is a particularly strong and important feature of Little-JIL. Steps can be preceded by a prerequisite check (indicated by a green triangle to the left of the step bar in Figure 1) and followed by a postrequisite check (indicated by a red triangle to the right of the step bar in Figure 1). Each of these can represent an entire step structure that evaluates to true or false. If a requisite evaluates to false then an exception is thrown. The agent for a step can also throw an exception during the execution of the step. Exceptions are typed objects, and are handled by handlers for that exception type that are above the step that has thrown the exception in the step decomposition hierarchy. Every step can contain one or more exception handlers, each of which may itself be an entire step hierarchy. A step's exception handlers are attached to the step by edges that emanate from the right side of the step bar. When a step contains exception handlers, a red X appears inside the right side of the step bar. Non-leaf steps are sometimes introduced into the step hierarchy specifically for the purpose of creating a scope of applicability for a particular exception handler. Little-JIL's exception management facilities seem particularly well suited to support the clear and concise specification of various kinds of iteration that must occur in realistic formal verification activities.

3.2 A Little-JIL Definition of Floyd's Method

Figure 2 depicts a Little-JIL definition of Floyd's Method. As might be expected, the process consists, at the highest level of abstraction (represented by the substeps of the root step in Figure 2), of sequential execution (indicated by the right-facing arrow in *Floyds Method,* the root step) of *Define and Place Initial and Final Assertions*, a step to create and place all needed assertions, then *Define and Place All Invariants*, a step to create all of the needed loop-breaker invariants, *Create Lemmas*, a step to build all of the lemmas whose proofs are needed to complete the formal verification of the program, *Prove Lemmas*, a step to actually carry out the proofs of all of the lemmas, and then *Prove Program Termination*, a step to prove that program execution must terminate.

While these top level steps capture the nominal process of performing Floyd's Method, our process is designed to support the actual process that a verifier most typically goes through, including the recovery from errors and speculative approaches that

do not work out, and supporting verifiers' efforts to determine when such recoveries are need, and how to carry out the recoveries. In the lower levels of this process definition we will show some examples of such error detection and recovery. At the top level, though, we can already see such an example. Hanging from the right end of the step bar of the root step, *Floyds Method*, is an exception handler that shows that the entire *Floyds Method* process will be reexecuted in response to an appropriately typed exception thrown at any time during the execution of the process. This presumably would happen in response to a realization that the execution of the process has become hopelessly entangled (we shall see shortly how easily this can happen). In this case the process definition specifies that the hopelessly entangled verification process execution will be abandoned and the entire *Floyds Method* process will be restarted. We note that the reinvocation of *Floyds Method* is essentially a recursion, indicating that the new process invocation takes place in the context created by this exception handler, where that context (communicated perhaps through arguments thrown with the exception) may incorporate important information about what has been tried previously and why it has not worked out well. Other examples of exception management will be shown in the elaboration of the top level steps that we address now.

To provide further examples of the power of our process definition to describe and guide the details of a realistic performance of Floyd's Method, we now describe elaborations of some top level steps, starting with *Define and Place All Invariants*. This step is an iteration over all of the invariants needed to support the complete proof.

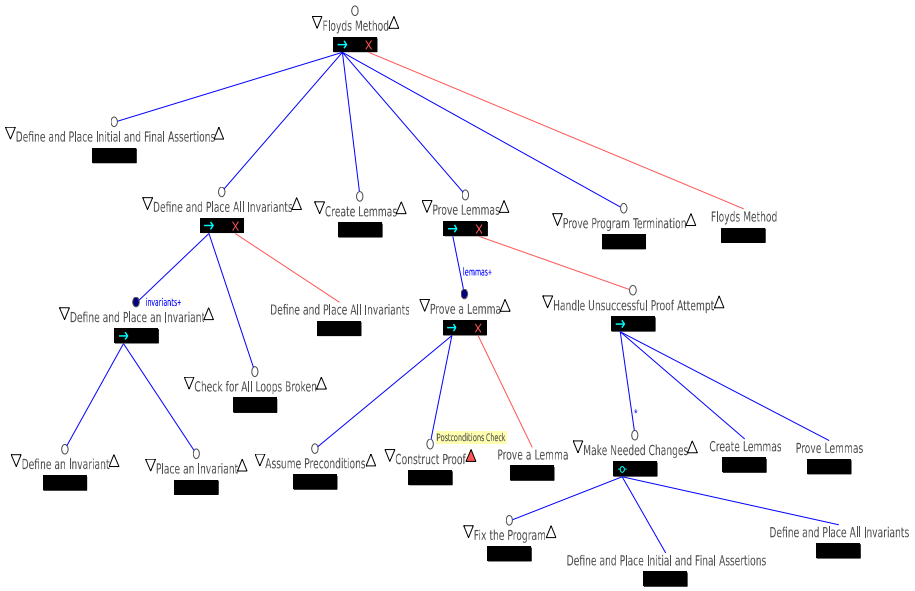

Fig. 2. An Example of a Little-JIL definition of a process for performing Floyd's Method of Inductive Assertions

The iteration is indicated by the annotation *invariants+* positioned on the edge between *Define and Place All Invariants* and *Define and Place an Invariant.* The *invariants+* annotation on the edge indicates that the lower level step is to be instantiated one or more times (the Kleene +), once for each invariant that the verifier wishes to create. The lower level step, *Define and Place an Invariant*, is decomposed into two further substeps, namely the defining of the invariant and the placement of the invariant in the program source text. Following performance of the iterative definition and placement of all invariants, is a substep of *Define and Place All Invariants, Check for All Loops Broken*, that is imbedded in the process here to assure that enough invariants have been created and that they have been appropriately placed. It is expected that this step is to be executed by some kind of automated tool, and that can be specified as part of the Little-JIL definition of this process, although this visualization of the definition does not include that annotation in order to reduce visual clutter. If the checking to assure that all loops are broken by an assertion reveals that this critical requirement has not been met, then the checker will throw an appropriately typed exception. This exception is to be handled by the *Define and Place All Invariants* step, which consists of reinvoking *Define and Place All Invariants*. Here too, this is a recursive invocation of the step in which the new execution context will presumably contain information about the cause of the exception, most likely a specification of all of the loops that have not yet been broken. This information would presumably be of considerable value to the verifier in identifying just where additional assertions are needed. As the entire *Define and Place All Invariants* step is reinstantiated, however, the verifier will in this case also be free to edit, remove, or replace any existing assertions. The reinstantiations of the *Define and Place All Invariants* step will continue until checking confirms that all loops have been broken.

The elaboration of the *Prove Lemmas* step incorporates other interesting details. *Prove Lemmas* is also an iteration whose nature is specified by the *lemmas+* annotation on the edge from *Prove Lemmas* to its substep, *Prove a Lemma*, which will be instantiated once for each of the lemmas to be proven. The list of lemmas to be proven will have been generated by *Create Lemmas*, the immediate predecessor sibling of *Prove Lemmas*, and then passed as an argument from *Create Lemmas* to *Prove Lemmas*. *Prove a Lemma* consists of the sequential execution of *Assume Preconditions* and then *Construct Proof.* Certainly most of the time and effort in verification is spent in the *Construct Proof* step, which we do not elaborate here. This is the heart of program verification, and is often a highly creative activity. Some guidance in how to carry out that activity could be provided by elaborations of this Little-JIL step, perhaps emphasizing the details of how humans and proof assistants might collaborate. Thus, further elaboration of this step would seem to be particularly important if this process is to be used to train and support formal verification novices. Elaborations of this kind will be pursued in future work on this ongoing project.

To see how this process definition does help support assurance about the correctness of the proof, note that the *Construct Proof* step has a postrequisite, presumably (but not necessarily) carried out by an automated proof checker, whose job is to confirm that what has been created is indeed a valid proof of the lemma. If the postrequisite determines that the proof is not valid, then two different exceptions might be

thrown. One exception, suggesting a defect in the proof, would be handled by the *Prove a Lemma* step, and would cause the *Prove the Lemma* step to be reinvoked recursively thereby causing the verifier to examine the report of the defect in the proof thereby expediting the process of correcting the defect.

The postrequisite in this definition is also able to throw a different exception, this one to be handled by *Handle Unsuccessful Proof Attempt*, an exception handler attached to the higher level *Prove Lemmas* step. *Handle Unsuccessful Proof Attempt* is to be thrown when a failed proof attempt has indicated that there is a more fundamental problem with the verification. It is not uncommon, for example, for an assertion to have been incorrectly stated, or inconveniently placed, or indeed for the program that is being verified to contain a defect (a principal reason for doing the verification). Each of these three possible difficulties is to be addressed by a different substep of *Make Needed Changes*, the first substep of *Handle Unsuccessful Proof Attempt*. *Make Needed Changes*, is defined to be a choice step, indicating that the verifier is free to choose whichever of its three substeps (each supporting a different kind of change) is to be performed. Of particular note is the fact that the edge from *Handle Unsuccessful Proof Attempt* to *Make Needed Changes*, is annotated with a Kleene *, indicating that this step can be instantiated as many times as the verifier might wish, in order to support the possibility that the verifier may need to make more than one change (e.g. perhaps to modify more than one assertion, and perhaps also change some program source text). Regardless of the number of instantiations of *Make Needed Changes*, the process next mandates that *Create Lemmas* and *Prove Lemmas* be executed next. The reexecution of *Create Lemmas* is to assure that all lemmas continue to be consistent with the current set of loop-breaker assertions (some of which might have been changed during the execution of *Make Needed Changes*). The reexecution of *Prove Lemmas* requires that all lemmas be proven. In cases where neither the assertions nor code have been changed, the proof would not need to be modified, and the postcondition on *Construct Proof* would reconfirm the validity of the previous proof. In other cases, *Prove a Lemma* would entail creating a new proof, and would require the successful execution of the *Construct Proof* postcondition. Failure of the postcondition would cause the throwing of an exception and might cause another recursive invocation of *Prove Lemmas*.

It is clear that continued recursive invocations of this step, perhaps multiple times in response to multiple postcondition failures, can create a situation in which several different proof failures are being investigated and corrected essentially simultaneously, perhaps even in ways where progress in addressing one difficulty creates new difficulties for other proof correction activities. All of this creates a potentially very confusing environment for the human verifier. Little-JIL's hierarchical structure, and its ability to support recursive invocations, each of which can carry considerable amounts of contextual information, seems to have the clear potential to provide guidance that could be very useful to the verifier in this sort of complex situation. Still, however, it is not hard to envisage situations in which there are so many recursive reconsiderations of so many lemmas that the verifier might feel it is best to simply start all over again. In that case, this process definition supports the ability of the human verifier to throw the exception this is handled by the top level *Floyds Method*

step, causing the entire process to be started anew (although still in a context that might make information about the previous verification attempts available).

4 Analysis of a Formal Verification Processes

As noted above, we feel that it is not sufficient only to verify a program, but that it is also important to be able to provide to stakeholders (including the human verifier—especially a novice human verifier) credible evidence that the program has indeed been verified, and verified correctly. In this section we demonstrate how a formal definition of a formal verification process such as the one just presented can be effective in supporting the creation of such evidence. To provide some initial examples, we suggest some fundamental assurances are that: all loops have been shown to have been broken by at least one assertion, that all necessary lemmas have been created, that each lemma is stated based upon the most recently created assertions and current code, and that each lemma that has been verified is indeed a lemma based upon current assertions and code. The nominal execution of Floyd's Method as a straight uniterated sequence of the top level substeps of *Floyds Method* makes it clear that a verification cannot terminate until and unless all lemmas have been created and proven. But experience suggests that the verification of a real program is almost never as simple as following that nominal uniterated straight path, because errors in creating assertions, placing them, creating lemmas and (especially) proving lemmas successfully are to be expected, necessitating backtracking, and iterations of various kinds. As noted above, all of these difficulties are further complicated when reverifying a program that has been modified. In such a case some assertions, lemmas, and proofs can be reused, but others cannot. We suggest that proactive process support can be quite useful both to verifiers and other stakeholders, in providing guidance about which assertions, lemmas, and proofs are safe to reuse and which are not. The example process presented above is a suggestion of one way in which needed backtracking and iteration might be organized systematically, hopefully providing structure and artful integration of automated tools that can be of real assistance to a human verifier, especially a novice human verifier. But this process definition also makes it clear that the specification of backtracking and iteration also make it more difficult to be sure that all needed lemmas have been created and then proven successfully, with no changes to any assertions or program text taking place between the conclusion of all of these proofs and the end of the execution of this process.

In earlier work we have argued that a process that is defined in a rigorously specified language can be thought of as a kind of software that is, in particular, amenable to analysis using existing software analysis approaches, such as testing, static analysis, and formal verification [9,10]. That suggests to us that it should be possible to apply these kinds of analyses to formally defined formal verification process definitions such as the one presented above in order to produce assurances to stakeholders of the correctness, and the correct execution, of such processes.

We begin by observing that the executability of a Little-JIL process supports the ability to do dynamic analyses of executions of the process. Little-JIL processes are

executed using a system, Juliette [14], that assures that steps are executed in orders that are consistent with Little-JIL semantics, and that arguments are correctly collected from completed steps and delivered to steps that are defined to be their users. Juliette also delivers to agents, both human and automated, the work that they have been assigned as per the specifications in the process definition. These work assignments are delivered to the agendas of the agents, who signal the completion of assigned work by making appropriate annotations to their agendas. Clearly the history of inputs received, assignments of steps to agents, the completion statuses of those steps, and the values of artifacts both consumed and generated, comprises an articulate record of how the process has been executed for any given input program. This record, generated in the form of a Data Derivation Graph (DDG) [11], seems to be an excellent basis for reasoning about whether key properties have been adhered to by any execution of the process. Thus, for example, if we wish to be sure that all lemmas have been proven successfully and correctly, we should be able to verify this by examining the DDG to see that no changes have been made either to a lemma's assertions, or to the lemma's code subsequent to the last successful verification of that lemma, and prior to the completion of the execution of the entire process. By confirming that this is indeed the case for each lemma, we then confirm that all of the component parts of the verification have been proven. If the postcondition of the *Construct Proof* step has been executed by an automated proof checker, then this will be observable from an examination of the DDG, and reportable as part of the assurances provided to stakeholders.

As is the case with other kinds of software, however, this sort of dynamic analysis of a single formal verification process execution may provide useful assurance about the trustworthiness of a single verification, but provides no assurances about any other verification process execution. More generally, in addition to knowing that a single verification has satisfied some key properties, it is also quite important to know that any verification that follows the same process must also always satisfy these key properties. Thus, we believe it is important to be able to analyze a given formal verification process definition to verify properties such as that all lemmas have been proven correctly prior to the termination of the verification process. This can be done by applying finite state verification [3] to a rigorously defined verification process specification such as the one just presented. As an example, we indicate how this property can be verified for the specific process shown in Figure 2.

A finite state verification of the property that all lemmas have been successfully proven before the process terminates should be based upon analysis of a flow graph derived from the Little-JIL process definition. Because Little-JIL's semantics are rigorously defined based upon finite state machines, such a flowgraph can be automatically generated (and indeed we have developed a tool for automatically generating such process definitions into the Bandera intermediate representation, which incorporates all needed data and control flow information [3]). The verification of this property then entails verifying that no modifications to a lemma's statement or to the program text used by the lemma can occur between the end node of the process flowgraph and the most immediately proximate prior successful execution of the proof checker's confirmation of the correctness of the proof. An informal inspection of the

process definition strongly supports a surmise that this is the case, as the termination of the execution of the process occurs only after the successful completion of the *Prove Lemmas* step (regardless of how many times this step might have been called recursively), and the successful completion of the *Prove Lemmas* step occurs only after all of the proofs of the individual lemmas have been confirmed as being correct.

Certainly, however, this informal argument should be supplanted by a rigorous proof of the consistency of the process with this property. To cite a specific suggestive reason, suppose that the *Create Lemmas* step had been omitted from the list of children of the *Handle Unsuccessful Proof Attempt* exception handler step. If that were the case, it might then be possible that the *Define and Place All Invariants* step might have been performed as part of the response to an unsuccessful proof attempt, thereby potentially necessitating the modification of some prior lemma statements. But, with the *Create Lemmas* step missing, the process now no longer assures that those new lemmas would be created, and thus does not assure that the needed new proofs would be completed. Determining whether the needed proofs have been correctly created and proven would have to be determined based upon a careful analysis that included precise specification of the flow of arguments between these process steps. Certainly this is eminently possible, based upon the semantics of Little-JIL. But the purpose of this thought-exercise is to suggest that just this kind of careful and precise mathematically sound static analysis of processes such as this one is necessary—but possible. Similar kinds of concerns about the soundness of a proof arise in being sure that a reverification has not relied upon lemmas that are no longer valid, or upon lemmas that have been incorrectly constructed using assertions that are no longer valid.

Indeed, going a step further, the kind of analysis that is probably of most value is a full formal verification of this formal verification process definition. We suggest that it is not only possible, but actually highly desirable, to apply Floyd's Method to the verification that this process for the performance of Floyd's Method necessarily always produces a correct outcome. Thus, for example, we would like to use Floyd's Method to verify formally that all lemmas needed for a verification have been created correctly, and that every one of these lemmas has been proven correctly. Doing this requires, obviously, the creation of at least one loop-breaking invariant assertion for each of the loops in our process definition. As in the case of the verification of any other kind of program, this requires a firm understanding of the essential goals and natures of each of the loops, which is perhaps the most valuable product of any verification effort. In the case of the iteration in the *Prove Lemmas* step, for example, a component of the needed invariant would certainly have to assert that at the end of the i^{th} iteration all lemmas up to and including $lemma_i$ have been proven successfully.

We have not, at present, undertaken the formal verification of a formal verification process, as this work is currently ongoing. But we expect that the somewhat intricate structure of exceptions and exception handling in this example verification process might make the proof of all needed lemmas difficult, perhaps suggesting the advisability of creating a more straightforwardly understandable and provable process. This is very much in line with what our community has learned about the value of

verifying programs, namely that proof attempts often lead to the kinds of deeper understandings that point the way to useful simplifications.

The fact that program verifications currently are often carried out (e.g. by novices or those having to work in environments that lack powerful proof assistant tools) without benefit of a formally defined process to guide them raises the risk that reported successes in verification may not be supportable by acceptably rigorous arguments and acceptably definitive evidence. The example we have just presented makes it clear that the expectable need to incorporate various kinds of iteration and rework into real verification processes can make it difficult to assemble such rigorous and definitive evidence. To address this worrisome problem this paper has suggested an approach that should seem quite natural to those who have developed the admirable science and technology of testing, analysis, and verification.

5 Conclusions

We have argued that it is quite important to be able to assure all stakeholders that a formal verification of a program has been carried out correctly. This seems particularly important in consideration of the fact that formal verifications typically entail extensive amounts of rework and iteration that can raise doubts about whether all necessary lemmas have really been generated and proven. We have then demonstrated that the process of performing just such an iteration-intensive formal verification can be defined precisely in a rigorously defined process definition language. We then indicated how classical dynamic, static, and formal verification approaches can be applied to such a process definition thereby creating the kinds of assurances of verification correctness that should be desired by stakeholders. But our work is still in a relatively early stage, and so in this paper we describe the application of these analysis approaches to only one specific desirable property. Future work must address far more properties.

In addition, this paper has suggested the applicability of only a few analysis approaches. But in future work it seems useful to consider how to apply other approaches. For example, dynamic monitoring could be used to help identify proof bottlenecks and sticking points and to suggest possible approaches to resolving such problems, perhaps aided by the results of automated proof assistants that might be automatically invoked. Moreover, timing analyses might be carried out as well to attempt to project the amount of time needed to complete a verification. Most important, perhaps, is this paper's suggestion that formal verification has applicability to more than just programs, but also is a valuable technology to apply to processes as well, even the processes that should be used to carry out formal verification.

Acknowledgments. The author wishes to thank Xiang Zhao for developing the Little-JIL definition of Floyd's Method shown as Figure 2 in this paper. This work described in this paper has been supported by the National Science Foundation under grants IIS-1239334, CNS-1258588, and IIS-0705772.

References

1. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
2. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
3. Dwyer, M.B., Clarke, L.A., Cobleigh, J.M., Naumovich, G.: Flow Analysis for Verifying Properties of Concurrent Software Systems. ACM Transactions on Software Engineering and Methodology 13(4), 359–430 (2004)
4. Floyd, R.W.: Assigning Meanings to Programs. In: Schwartz, J.T. (ed.) Proceedings of a Symposium on Applied Mathematics, vol. 19, pp. 19–31 (1967)
5. Futatsugi, K.: Verifying Specifications with Proof Scores in CafeOBJ. In: Intl. Conf. on Automated Software Engineering, Tokyo, Japan, pp. 3–10 (2006)
6. Futatsugi, K.: Fostering Proof Scores in CafeOBJ. In: Intl. Conference on Formal Engineering Methods, Shanghai, China, pp. 1–20 (2010)
7. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576–580 (1976)
8. King, J.C.: A Program Verifier., PhD Thesis, Carnegie Mellon University (1969)
9. Osterweil, L.J.: Software Processes are Software Too. In: ACM SIGSOFT/IEEE 9th International Conference on Software Engineering (ICSE 1987), Monterey, CA, pp. 2–13 (March 1987)
10. Osterweil, L.J.: Software Processes Are Software Too, Revisited. In: ACM SIGSOFT/IEEE 19th International Conference on Software Engineering (ICSE 1997), Boston, MA, pp. 540–548 (May 1997)
11. Osterweil, L.J., Clarke, L.A., Podorozhny, R., Wise, A., Boose, E., Ellison, A.M., Hadley, J.: Experience in Using a Process Language to Define Scientific Workflow and Generate Dataset Provenance. In: Proceedings of the ACM SIGSOFT 16th International Symposium on Foundations of Software Engineering, Atlanta, GA, pp. 319–329 (2008)
12. Paulson, L.C.: The foundation of a generic theorem prover. Journal of Automated Reasoning 5(3), 363–397 (1989)
13. Wise, A.: Little-JIL 1.5 Language Report. Department of Computer Science, University of Massachusetts, Amherst, UM-CS-2006-51 (2006)
14. Wise, A., Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J., Sutton Jr., S.M.: Using Little-JIL to coordinate agents in software engineering. In: Proceedings of the Automated Software Engineering Conference (ASE 2000), Grenoble, France (2000)