

A Formal Advisor for Eclipse Java Development

Frank Rimlinger
US Dept. of Defense
Fort Meade, MD
Email: frankrimlinger@gmail.com

Abstract—A “formal advisor” takes source code as input and produces proof artifacts for an automated theorem-prover as well as precise, human readable functional documentation. The formal advisor determines what the theorem-prover should prove and what useful information can be provided as code documentation. To be effective, such a system must be highly automated and integrated with a development environment. The paper describes an evolving formal advisor for Java code, driven by the JPF engine embedded within the Eclipse development platform.

I. INTRODUCTION

Java developers will often provide *javadoc* for some of the methods of a class. This documentation is usually a brief summary intended to apprise would be callers of the method of what inputs are expected and what outputs should result. If the documentation is good, the developer need not even review the source code to resolve crucial development issues. Does the method perform the desired function? Is the input correctly set up? Are any assumptions about the environment properly accounted for? Conversely, in the complete absence of source code and javadoc, the developer must just guess the intended function, drawing on experience, context, the name of the method, and any black box testing that time allows. Inevitably, errors creep in, and so it is natural to ask, how might this error rate be lowered?

Certainly the error rate would be lowered if all methods had good documentation in the above sense. The question then becomes, how to obtain this documentation? As no one sets out with the idea of writing incorrect code, informally generated documentation reflects developer intentions, which may or may not coincide with the function of the code as written. *Observational bias* may be defined as the erroneous identification of intent with function. Fortunately, there are ways to eliminate observational bias.

For example, consider obfuscating all identifiers in the code. The obfuscated code still determines the same function, and can now be read without observational bias. Unfortunately, obfuscated code is incomprehensible, and so clearly fails the test for good documentation. But this strategy contains a grain of truth and can be improved upon. Imagine a formal transformation of the code to an alternative description of function which is comprehensible, and yet sufficiently different from the original code that it may be read without observational bias. A *formal advisor* generates this description. This paper proposes a mechanism to drive a formal advisor, and reports on project *Mango*, an effort to build this mechanism [11].

II. FORMAL ADVISOR MECHANISM

In this section a series of steps is taken to modify the control flow diagram representing a computer program. Each such modification comes with an implied prescription for enumerating the execution paths through the new diagram. The net effect of program execution on each input state is equivalent to that of the original diagram, so that program function is invariant under these modifications. The point of this modification process is to break up control flow into a hierarchy of diagrams. This hierarchy is then analyzed in a bottom up manner via a uniform process. This method of analysis was first described in [12].

A. Loop subdivision

Abstractly, a control flow diagram is a finite graph whose vertices are state transition functions, or less formally, instructions, together with oriented edges which determine the flow of execution from one instruction to the next. Each such edge corresponds to a state predicate, sometimes called a *branch condition*, deciding which states may pass from edge source to edge destination. A sequence of edges then corresponds to an execution path if the destination of each edge is the source of the next. Accordingly, a minimal path that begins and ends at the same instruction corresponds to a loop in program execution, and a computer program contains a finite number of loops.

An instruction is a loop *entry point* if it is the destination of an edge not in the loop. A loop entry point I may be replaced by two new instructions, α and Ω , such that all edges formerly leaving I now leave α , and all edges formerly entering I now enter Ω (see Figures 1 and 2). Observe the paths from α to Ω in the new graph now determine a control flow diagram (see Figure 3). Let S represent the state transition of this diagram, and let C accept the states s entering the loop, such that $S^n(s)$ exits the loop for some number $n = n(s)$ which may depend on s . Formally, define the recursive function

$$F(s) = \begin{cases} s & \text{if } n(s) = 0, \\ F(S(s)) & \text{if } n(s) > 0. \end{cases} \quad (1)$$

This function F becomes the state transition for Ω , and α is given the identity state transition. The execution paths through the new diagram are as before, except that paths arriving at Ω go directly to α and are henceforth constrained never to arrive at Ω again (see Figure 4). Overall program execution remains

unchanged, but the complexity of the diagram is reduced by at least one loop.

Repeat this process until no loops remain. Observe that the benefit of simplifying the graph is evidently offset by the cost of introducing a hierarchy of recursive functions. However, this is not inconsistent with the stated goal of producing a *different* point of view, one from which we may observe functionality without observational bias. By analogy with signal analysis, time and frequency domains both present useful points of view, but neither is typically simpler than the other. In particular, the recursive functions generated by this process are suitable for analysis by automated theorem provers such as ACL2 [9][6].

B. Confluence

A vertex of a graph G is a *branch point* if it has at least two outgoing edges and a *confluence* if it has at least two incoming edges. The *dual graph* G_* is obtained by reversing the orientation of all edges of G , so that the branch points of G are the confluences of G_* , and $G_{**} = G$. Given branch points b_1 and b_2 in a graph, say that b_1 *flows into* b_2 if there is an execution path from b_1 to b_2 which contains no intermediate branch point. Observe that the branch points of a graph G themselves form a new graph $B(G)$, such that the source b_1 of each edge e of $B(G)$ flows into the destination vertex b_2 of e , as b_1 and b_2 are viewed from within G .

A graph G is *acyclic* if it contains no loop. For example, the ultimate control flow diagram obtained in the previous section is acyclic. Certainly if G is acyclic, then so are $B(G)$ and G_* . Given a finite acyclic graph G (see Figure 5), evidently $B(G)$ must contain a sink vertex b (see Figure 6). Define the *forward flow* G_b of b within G to be the subgraph of G supporting execution paths from b to sinks of G . Observe that G_b has exactly one branch point, namely b , although it may have many confluences.

Now repeat this argument, starting with the dual graph G_{b*} , arriving at a graph K with at most one branch point and at most one confluence (see Figure 7). The paths from the branch point to the confluence are *parallel cases*. Observe that K_* is now naturally embedded in G , and may be excised, reducing the number of branch points of G (see Figure 8). Observe that only the edges of K_* in the complement of the forward flow of the remaining branch points are removed from G .

Just as a loop may be replaced by a recursive function, a set of parallel cases may be replaced by a single case. This single case is the *unification* of the parallel cases. Unification is an algorithmic process which introduces variables in a generic expression and then provides a *specialization* for each particular case. Replace the parallel cases of K_* by a single unified case and sew the consequent edge back into G (see Figure 9). This process reduces the number of branch points of G , allowing the construction of the functional description to proceed without exponential growth in the number of execution paths. This is an absolute requirement for the procedure; however, it does not come without a cost, as we shall see in the next section.

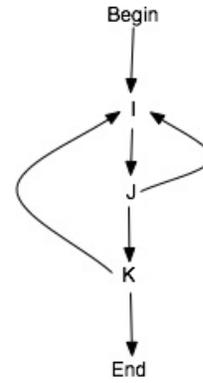


Fig. 1. Control flow with loops

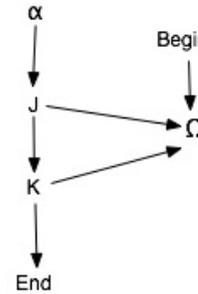


Fig. 2. Entry point I replaced by α and Ω

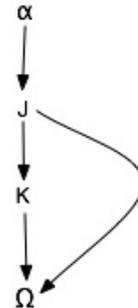


Fig. 3. The body of the recursive function F

C. Loop invariance

Each execution path determines a state transition S and a constraint C , which is a boolean function of state. Given a state s , the constraint C will accept s if and only if each successive transition of s by a path instruction is accepted by the corresponding outgoing edge branch condition. More succinctly, C decides which states will flow along the given path from beginning to end. Taken together, S and C form a *specification* for the path. Such specifications, suitably translated into natural language, comprise the model which



Fig. 4. F replaces the loop

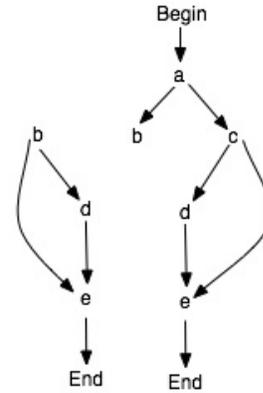


Fig. 8. Excision of K_* reduces complexity

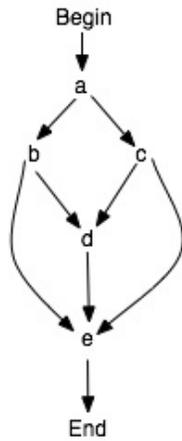


Fig. 5. An acyclic graph G

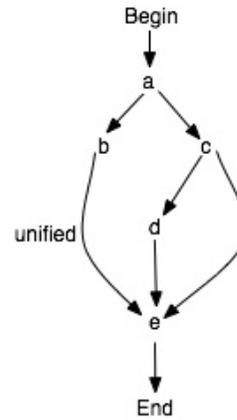


Fig. 9. Branch point reduction of G

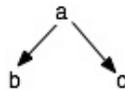


Fig. 6. The branch graph $B(G)$

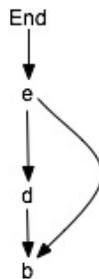


Fig. 7. K , the reduction to parallel cases

the formal advisor exposes to the user as the equivalent but different description of program function.

Before the matter of translation and exposition can be taken up, the actual computation of S and C must be considered. At this point, design choices must be made which have profound impact on the architecture and efficiency of the formal advisor. Specifically, a model of state must be chosen, together with a model for representation of state transitions and branch conditions. Taken together, these choices are sometimes referred to as *symbolic simulation*. For the purpose of this section, assume that such a symbolic simulation engine is in place. The question now becomes, how will this engine deal with the constructs introduced in the previous sections, namely, recursive functions and generic state transitions?

Given a recursive function F , it is always good to know for states s satisfying some input-time constraint C , if $F(s) = s$. In this event, C is said to be F -invariant, or speaking more loosely, C is a *loop invariant*. Efficient computation of loop invariants is essential for efficient computation of path specification. Of course, the general question of F -invariance

of a constraint C is undecidable, but in practice this is not an issue. The real question is how to efficiently store and retrieve the information required to attempt an invariance detection computation in the first place. The difficulty is compounded by the presence of generic input state, which may be present due to unification of parallel cases upstream of the loop. Observe that brute force loop body simulation of individual specializations quickly leads to exponential growth of computational complexity in the presence of deeply nested loops.

Fortunately, failure to detect a loop invariant does not lead to unsound results. However, the higher the success rate of invariant detection, the lower the amount of self-referential material ultimately exposed to the user by the advisor. For an example of the ambiguity introduced by self-reference, ask me what my name is. I could respond "My name is Frank", or perhaps "My name is that which identifies me". Both responses are correct, but the former is clearly far more informative than the latter. The successful loop invariance analyzer must therefore have a high success rate for typical situations, and be robust enough to abandon computations that create unacceptable computational load.

The approach taken to invariant detection in Mango is sound, but also both *ad hoc* and fragile. This part of the tool is still evolving and contributions in this area would be most welcome.

D. Translation

Expressions produced by the symbolic simulation engine within Mango are based on the state model of the Java Virtual Machine (JVM) [5]. Such expressions are suitable for automated processing, but far too complex and repetitive for human reading. This section discusses how such expressions can be exhibited and navigated in a human-friendly manner. See Figure 10 for some an example of the Mango plugin working within the Eclipse workbench. Please see [11] for examples of tool output. This site will be updated as the tool matures and the examples become more interesting.

State is modeled with five variables: H , nH , $Stack$, $Stat$, $oldState$, for the heap, next free heap address, frame stack, static area, and previous state, respectively. The *Mango Formal Language* (MFL) uses Lisp syntax [13], so, for example, a generic state looks like

```
(slist H nH Stack Stat oldState)
```

State transition is modeled at the byte code level, with uninterpreted *rule keys* modeling JVM semantics. For example

```
(getFromLocalVar 1 (topFrame Stack))
```

represents the value of the local variable at offset 1 in the top frame of the frame stack. Accordingly, the first step in translation is to map such expressions to the corresponding local variable names. For example, within the context of the method

```
static public foo(int x,int y){. . .}
```

the above expression would just become y , greatly improving readability. Of course, the introduction of context requires a management layer that handles context switching, but this theme will be taken up in section III. The next level of translation involves the conversion of lisp syntax into natural language. This is accomplished with a rule-based pattern matching system. For example, $(\text{not } (= y \text{ null}))$ matches on the rule " $(\text{not } (= x \text{ null})) \mapsto x \text{ is defined}$ " to produce " y is defined".

Computation of the specification (S, C) is incremental, essentially one byte-code per heartbeat. At each heartbeat, an attempt is made to simplify the set of accumulated branch conditions via automated reasoning, and to remove any recursive function invocations via loop invariant detection. A metric is imposed to measure the complexity of the updated (S, C) expression once all simplification has taken place. If the expression is too complex, simulation pauses and the user is alerted. At this point the user may explicitly introduce rules to reduce complexity. Typically, after a training period, this issue settles down.

Analysis proceeds bottom up, with the translation process in lock-step. Specifications for each generic case and each recursive function body are stored in persistent storage as built, so that the process handles interruption with minimal loss. Once all the components of a method have been assembled, the user may then query the overall specification of that method. The result is exposed to the user as a folder which may be opened to reveal OR folders, AND folders, and leaf expressions. The OR folders correspond to functions of generic state. If an OR folder is left unopened, the corresponding function will be rendered using only type reasoning based on the corresponding generic values. If an OR folder is opened, then the corresponding function cannot render until one of folder child items is selected. Each such child item corresponds to a specialization, and the specialized expression is then incorporated into the rendering logic. Observe that the specialization may itself contain zero, one, or more generic functions, and accordingly is a leaf, OR, or AND folder. Opening an AND folder has no effect on rendering, but does allow the user to navigate to the corresponding OR children.

In general, this exposition model exported to the user may be a *Pandora's Box*, in that the user typically must open an exponentially growing number of folders to observe all cases. However, for sufficiently well-designed, correct, modular code, the user should not need to drill down more than one or two levels to gain confidence that the descriptions obtained do in fact characterize the intended code function. Conversely, a failure to do so is generally indicative of some shortcoming within the code.

III. SYSTEM INTEGRATION

The JPF [10] project of NASA/AMES hosts the fundamental jpf-core [8] project as well as numerous project extensions, one of which is jpf-mango [11]. The jpf-mango site hosts an earlier stand-alone project constituting the original port from an even older C++ project, as well as the current Mango project. The

jpf-core project is used by Mango for two crucial functions. Parsing the Java `.class` files, jpf-core drives a notification based mechanism which can feed byte-codes in execution order to third party listeners. More generally, jpf-core can traverse any suitably configured graph, keeping track of branch points and *backing up* to them in a highly configurable manner. Mango uses both the specific service tied to Java for initial processing and the more general abstraction layer to traverse its own internally generated graphs.

Mango is intended to provide developers with near real-time feedback on their code as it is developed. As such, it must be part of a development platform. For this purpose, Eclipse [1] is an ideal platform for Mango. Deployed as an Eclipse plugin, Mango can be freely accessed by Java developers world-wide. Modern development projects typically consist of numerous files which the development platform maintains and exposes to the user in a coherent manner. Under the hood, Mango has access to all the structure maintained by Eclipse. This allows Mango to boot-up and organize its own data structures transparently, so that the user may begin code specification with just a few mouse-clicks. Mango also takes advantage of folder-viewer, graph-viewer, editor, view, marker, and persistent storage functionality supplied by Eclipse to bring the user an intuitive experience consistent with familiar Eclipse workflow.

A. Integration with jpf-core

Within the Mango project, the code for integration with jpf-core is contained within the `jpf - mango - bridge` source root. The `gov.nasa.jpf` package contains a few classes that handle JPF boot up issues. The bulk of the code is in the `mango` package, split between sub-packages `scanner` and `jpfmango`. The `scanner` package handles the initial parse of targeted `.class` files. Within `scanner`, the `bytecode` sub-package contains a class for each byte code. Guided by the specification for the Java Virtual Machine [5], each such class translates the byte code semantics into MFL. By this means, the state transitions S and the branch conditions C get built on-the-fly as the scanner runs. In addition, during the scanner run, the corresponding control flow diagram structure is built.

After scanning is complete, loop subdivision takes place, creating a hierarchy of blown-up loop components. This process is managed from class `LoopAlgorithm` in the `mango.worker.mangoModel` package of the `Mango.src` root. The `jpfmango` driver now takes over, with four nested levels of choice generation [7]. At the top level, the component dependency graph is walked. Each component G therefore is an abstract *instruction* which is *executed* by creating its corresponding branch point graph $B(G)$, and passing to the next level. At this level, the branch point graph is traversed bottom up, and each branch point b is executed by excising the forward flow G_b , passing to the dual G_{b^*} , creating the branch graph $B(G_{b^*})$, and passing to the next level. Similarly, $B(G_{b^*})$ is walked, at each point creating a parallel case graph K and passing to the next level. At the final level, K_* is traversed, and the mango heartbeat is executed for each maximal linear

path of byte-code instructions. The code for these levels is in the `jpfmango` subpackages `component`, `confluence` (both intermediate levels), and `caseSplit`, respectively.

The single greatest advantage of using the jpf-core to drive graph traversal is the emergence of a uniform concept of *time*. Regardless of the actual computation, the parcelling of code into the various listeners that occurs at salient times during traversal makes the code easier to understand and maintain.

The heartbeat work is controlled from the class `FreeChoice` and its extension `FreeInvocationChoice`. Invocation involves a lot of subtle context switching issues to maintain correct translation of local variables and specializations. The simplification, translation, and invariant detection processes are handled by the same basic pattern matching engine, referred to as the *rewriter*. The rewriter builds a substitution map $\{\text{variable} \mapsto \text{value}\}$ based on the pattern match of an input template with an expression. Classically, the substitution map is applied to an output template to produce the output expression. The Mango mechanism is generalized so that any *action* extending `mango.worker.engine.rule.RuleAction` may accept the substitution map and produce customized output. In sophisticated cases such code recursively invokes the rewriter. By this means, conditional rewriting, translation, and simple reasoning systems are layered on top of the rewriter. These activities are mediated by a `MangoThreadsOwner` in the `mango` package of the `src` root, which exposes *Pause* and *Cancel* buttons to the user.

When paused, the user may inspect various kinds of data, and add new rules to the rule base. The current jpf-mango [11] project may be pulled from the mercurial repository to develop new actions. The user and/or Mango developer may exercise the rewriter via a sandbox `MangoThreadsOwner` for the purpose of trying out new rules or actions or just generally trying to understand how expressions simplify. If the sandbox is active while specification generation is paused, then the user has access to the current constraint C accumulating along the path under specification.

The introduction of new rules and actions is strictly on the honor system. Aside from syntax checking and limited sanity testing, Mango has no ability to enforce its own soundness. Therefore it is always a good idea to perform lots of regression testing after altering the rule base or the code base. Currently there is no systematic way to do this, but this feature is definitely high on the list of things to do.

B. Integration with Eclipse

The controller for all the above activities is class `SelectTargetMsg`, in the `mango.worker.msg` package under the `Mango.src` root. This message is fired from the Eclipse gui, specifically the “Mango>Run” command within the Java editor window context pop-up menu of the user selected method. But before the run command may fire, the *method population* must be generated. Population generation is fired by the “Mango>Populate” command, also targeting the selected method. At this point, Eclipse internals are queried to determine what user project this method is in. Mango

takes the point of view that only source code in the user project need be specified. All other code dependencies are considered to be *native*, and must be either previously specified or provided with native specification. Eclipse internals are then used to generate all the source and native dependencies of the targeted method. This is the method population, which might be enormous. For this reason, computation of the method population is broken out as a separate command, so that the user may cancel and restart with a less dependent method if necessary.

Upon completion, the “Mango>Populate” command exposes the rule base and the existing set of specifications as folders in the “Mango Explorer” folder viewer (see Figure 10), analogous to the “Package Explorer” familiar to Eclipse Java developers. The user may navigate to a rule, for example, and pull it up in an editor window by double-clicking. All the familiar commands for editing, creating, copying, destroying, and moving rules about in the rule base are implemented. MFL expressions, such as those used within the rule pattern, substitution, and hypothesis fields, may be pasted into an “Expression Editor”. Each such editor comes with its own “Rewrite” command, together with “Pause” and “Cancel” buttons which are wired to the sandbox `MangoThreadsOwner`.

As specification proceeds, markers appear in the gutter of the source code editor of the targeted method to indicate specification progress. For example, in Figure 10, the red dot to the left of the `for` loop indicates the specification for the corresponding loop body has built. Clicking on such a marker creates a folder in the “Mango Explorer” for specification query. In Figure 10, this folder is represented by the green icon. The user may then inspect the specification further by opening this folder to reveal AND/OR children representing particular specializations. This representation is discussed in II.D above. A command may be fired at any time to render the current folder configuration as a specialized specification. The example of Figure 10 has no specializations, so just clicking on the green icon yields the complete specification as shown.

In addition to the specification level “Pause” and “Cancel” buttons, the “Mango Explorer” tool bar also exposes commands for “Reset” and “Synchronization”. In general, the rewriter uses a hashing mechanism to efficiently store expressions. The base class for this mechanism is `Hash`, in the `mango.worker.engine.hash` package of the `Mango.src` root. As the `jpfmango` mechanism backs up and generates new choices, so a family tree of hash tables is generated. Each expression knows its place in this family tree, and therefore whether or not it is stale at any given point in time. Stale expressions no longer have any interpretation in the current Mango model, and are removed from the “Mango Explorer” view by the “Reset” command. This eliminates clutter as the specification process posts a lot of time-sensitive data to assist rule-making and Mango development.

Necessarily, native specifications are editable, but in general specifications are read only. However, the user may use the “Package Explorer” to delete existing specifications for regression testing. The “Synchronization” command will cause

the “Mango Explorer” to completely rebuild itself to reflect the new state of the file system. This is just a stop-gap as Eclipse itself provides a comprehensive mechanism for auto-synchronization. For example, currently changes in the source code of specified methods do not invalidate corresponding Mango specifications, but this is something an Eclipse user would expect to happen automatically.

C. Integration with ACL2

Mango automatically generates *conjectures* about loop termination by asserting the truth of loop exit conditions evaluated on recursive function output. In order to prove these conjectures, loop body state transitions are translated into recursive function definitions which may be admitted into the ACL2 theorem prover logic. In addition to recursive functions, supporting definitions and other dependencies are also supplied. These output text files are suitable for processing by ACL2s [2], which is an Eclipse plugin supporting all ACL2 functionality within the Eclipse workbench setting.

The pipeline to ACL2s has been implemented in the `ac12` package in the `Mango.src` root. Robert Bellarmine Krug of University of Texas at Austin has helped with this modeling effort and accomplished complex termination proofs within ACL2 proper. However, this whole subject is in its infancy and lots of work remains. In particular, the issue of guessing induction metrics for loop termination needs to be addressed. For example, David Greve of Rockwell Collins has a paper [4] that might be a good starting point for this effort. Less sophisticated methods to just pattern match on typical situations and guess appropriate termination hypothesis are already in the pipeline.

The `TrailManager` class in package `mango.jpfmango.component` of the `jpf - mango - bridge` source root provides basic support for these efforts, identifying the *good* and *bad* exits from a loop. The idea is to prove that a good exit is attained, assuming the denial of the path constraint for the bad exits. The `FreeChoice` class notes the arrival at a bad exit, so that specification along the path can be cancelled, but currently no further action is taken. As a good loop exit edge is traversed, the corresponding branch condition composes with the loop function and so forms the loop termination conjecture. Much work remains to automatically generate suitable hypotheses for such conjectures, but the user does have the option of manually adding conjectures via *shadow rules*, which are rules that only take effect within the forward flow of a particular instruction in a particular specification environment.

IV. WORK IN PROGRESS

Currently, native methods must be specified by generating the requisite MFL expression by hand. This is far to labor-intensive. Automatic specification of native methods using *opaque* values is in the works. The heart of Mango currently beats several times per second, which is approaching near real-time. Planned improvements in the parametrization mechanism as well as in the invariant factorization data pipeline are

expected to further improve performance. Rudimentary auto-generation of loop termination hypotheses is on the drawing board.

Mango currently uses several sets of examples for road-testing. These are contained in the MangoHome and MangoSystem projects supplied by the plugin itself, also available for download from the top Mango wiki page [11]. Weaknesses in the tool are readily exposed by testing, but robust solutions often require several generations of design and testing to evolve. Once the tool becomes usable, it seems likely programmers would be willing to give it a try, lured by the promise of a human readable functional description. If a substantial user base does develop, then it will become possible to objectively measure whether or not the tool actually contributes to a reduction in programming errors.

V. RELATED WORK

The work of Max Ernst et. al. [3] on the Daikon system is particularly relevant for the future development of Mango. Once a candidate for invariance has been proposed, Mango can check invariance via symbolic simulation. Mango can store information required for checking in persistent form and utilize known loop invariants for simplification of downstream expressions. However, the current logic within Mango for actually finding invariants is weak. Rather than reinvent the wheel, it would appear to be a really good idea to just integrate the Daikon functionality for detection of likely invariants within Mango.

ACKNOWLEDGMENT

The author would like to thank Peter Mehltitz for general advice and for technical support on all JPF matters, in particular for advice on how to write the MangoThreads package, and Robert Bellarmine Krug at University of Texas, Austin, who developed the ACL2 model for Mango. Many thanks to those who helped bring Mango into the public domain, as well as all those who worked on the precursor to Mango.

REFERENCES

- [1] Eric Clayberg, Dan Rubel, *Eclipse Plug-ins, Third Edition* Addison-Wesley, January 2009
- [2] Peter Dillinger, Harsh Raju Chamarthi, *ACL2s, The ACL2 Sedan for Eclipse*, <http://acl2s.ccs.neu.edu/acl2s/doc/>
- [3] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, Chen Xiao *The Daikon system for dynamic detection of likely invariants* MIT Computer Science and Artificial Intelligence Lab, 32 Vassar Street, Cambridge, MA 02139, USA <http://www.cs.washington.edu/homes/mernst/pubs/daikon-tool-scp2007.pdf>
- [4] David Greve, *Assuming Termination* ACL2 Workshop Proceedings, 2009 <http://www.cs.utexas.edu/~sandip/acl2-09/final/22/22.pdf>
- [5] Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification, Second Edition* Addison-Wesley 1999 http://java.sun.com/docs/books/jvms/second/_edition/html/VMSpecTOC.doc.html
- [6] Hanbing Liu, *JVM models in ACL2*, <http://www.cs.utexas.edu/users/hbl/talk/JVM-models-in-ACL2.ppt>
- [7] Peter Mehltitz, *Choice Generators* <http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/choicegenerator>
- [8] Peter Mehltitz, **jpf-core**, *Basis for all JPF projects*, <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-core>
- [9] J Strother Moor, Matt Kaufmann, *ACL2 theorem prover*, www.cs.utexas.edu/users/moor/acl2

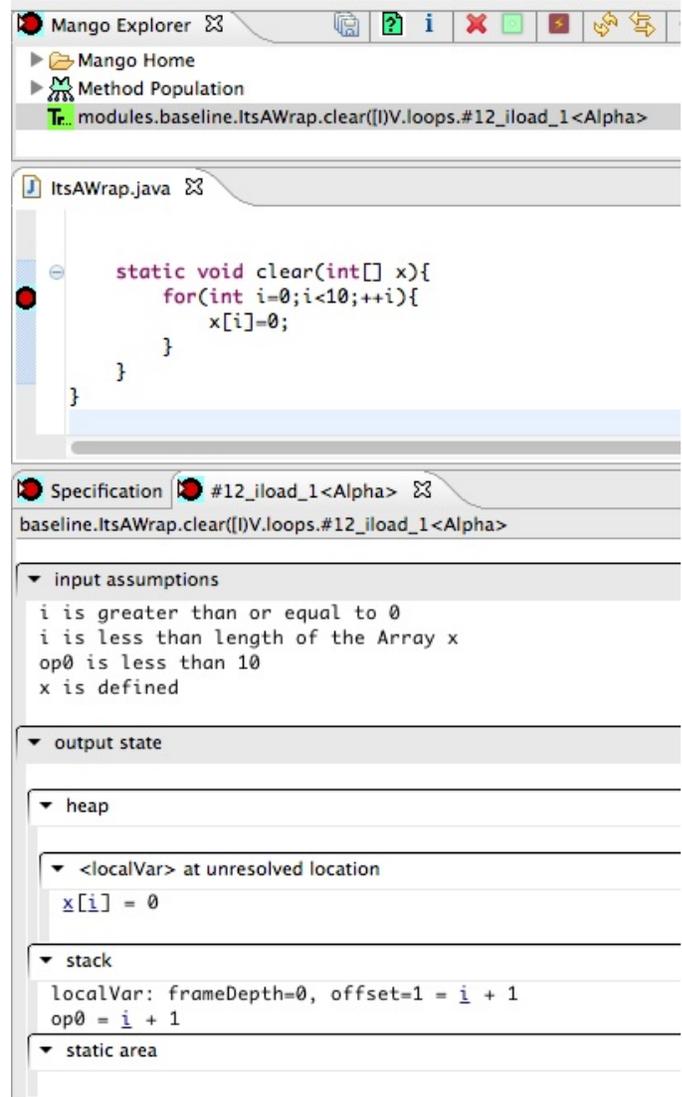


Fig. 10. Sample Mango output within Eclipse platform

- [10] NASA/Ames open-source **JPF** project, *The Swiss Army Knife of Java(TM) verification*, <http://babelfish.arc.nasa.gov/trac/jpf>
- [11] Frank Rimlinger, **jpf-mango**, *Specification and Proof Artifact Generation*, <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-mango> (wiki) <http://babelfish.arc.nasa.gov/hg/jpf/jpf-mango> (mercurial repository)
- [12] Frank Rimlinger, *Method of converting computer program with loops to one without loops* US Patent 7,788,659 B1, August 31, 2010
- [13] Guy L. Steele, *Common Lisp, the Language* Thinking Machines, Inc. Digital Press 1990 <http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>