

EclipseCon 2011 Slides

There are two sets of slides (code <--> model) which you can download. Just follow the attachments link at the bottom of this page.

Transcript

Hello, I am Frank Rimlinger from the National Security Agency, and I am going to talk about Mango. This tool generates proof artifacts for formal verification as well as a javadoc level description for the user. So far the tool has only been applied to small java code passages. The idea is to scale up by enabling greater interaction between tool and user.

Mango currently exists as a rich client platform application, part of the open-source Java Pathfinder project of NASA/AMES. The original technology was transferred a few years ago from the National Security Agency. The underlying mango model benefitted greatly from the assimilation of jpf core technology, but now it is time to shift gears and reconsider the user experience.

I have been working in software assurance for about ten years. Most of the time was spent trying to come up with a good model which can serve the needs of both formal verification and end-user experience. What I have today is the mango model, which is good, but needs work. Over time I realized that the context and control capability the user needs to understand and interact with the mango model is already available in the Eclipse Workbench. It's just a matter of swapping in a new brain.

To gain an appreciation of the issues involved in this port, let's look at a series of examples, starting with an old friend.

Here is the familiar Hello World example. Now I am going to bring up the mango output which describes this program at the javadoc level. In the world of functional programming, we think in terms of the input state, the function, and the output state. The function accepts the input state, suitably constrained by assumptions, and produces the output state. Mango renders the assumptions in brown, and the output state in blue. In this case, the assumption is not something you would normally worry about, but Mango isn't taking any chances.

In the next example, the clear method accepts an array of integers and zeros it over a certain range. Mango has a lot to say about this. Mango starts with the body of the loop, that is, the line $x[i]=0$. In brown you see the assumptions to remain in the loop and perform the array update, and in blue the result of a such an update. Notice that if i is not in range, or x is not defined, you will throw an exception, so these possibilities are assumed away. Now lets consider the loop from the point of view of the clear method.

We want the loop to terminate nicely, not because it threw an exception. Mango generates the good termination condition in red. Notice the hat sign, which means the condition is a function of loop output state, that is, a conjecture. The assumptions on the clear method input state which guarantee this conjecture are shown in green. These green assumptions deny the possibility of a thrown exception, and so imply the conjecture. Ideally, we would like to generate the green assumptions from the brown assumptions automatically for every program that ever was, but this is the well-known Halting Problem, which is logically impossible to solve. But in many typical situations, and certainly in this one, solutions are available. Currently Mango requires the user to examine the situation and guess suitable assumptions, but automation is in the pipeline.

Now lets look at a larger context, where a main routine calls clear, and then tests to see if $x[5]==x[6]$. Well, of course this is true since both values are zero, but how to prove it? In the main routine, mango generates the conjecture $x[5]==x[6]$. Since the user, or some automated decision procedure, had to guess the input assumptions, how do we know that they are correct? Well, we have to prove that the green assumptions imply the red conjecture. That's where automated theorem proving comes in.

In the picture, we see the proof of this conjecture generated by the ACL2 theorem prover of the University of Texas at Austin. In this case, the procedure was fully automated, but things don't always go so smoothly. If the theorem prover needs guidance, expert assistance is required. By this means, the highest level of assurance may be achieved. If there is no expert on hand to help with the proofs, Mango will gladly accept the truth of the conjectures anyway. One of the goals of the Mango project is to sort out what issues really require an expert and what cost-effective level of assurance can be achieved with just the programmer.

Let's move on to case splits. In the [CarRecall](#) example, we are confronted with serial true-false decision points.

Working from the end of the loop backwards, the first such decision point is the equivalence of modelNum and 0, then modelNum and 780, then 221, and so on. Here is a picture of what can go wrong when this program is approached on a case by case basis. If the case enumeration algorithm is sufficiently brain-dead, it is possible to generate an exponentially growing number of cases, which is a non-starter. Mango solves this problem by developing a hierarchical system of parallel cases. Lets take a look at the innermost case. What we are looking at here is internal to the mango model, not something exposed to the user. It is the control flow digram from the first decision point to the end of the loop, source in black, bytecodes in red, potential loop exit points in yellow. There are two parallel cases, only one of which mango will use, since the other will break out of the loop. The next set of parallel cases now references the first set, this is the trick that allows us to avoid exponential growth. The next set is similar, and we now have linear case growth at the price of less explicit description. Let's see how this plays out in the user output. Notice only one survivor in the first set, and the second set introduces variables to generalize the case description. In the third set, we see how the variables then refer again to variables of a previous set. Clearly this is a situation that cries out for hyper-link navigation.

But wait, it gets worse! What happens when we combine deeply nested loops with case splits? This happens, for example, in a program to play tic-tac-toe. Here is a picture of corresponding mango internal state. The funny red icons represent loops, and as you can see, there is a lot of nesting. Clearly we are going to need help from a powerful platform like the Eclipse Workbench to make sense of the mango model generated for tic-tac-toe.

Here are the marching orders for the port to the workbench. The basic idea is that with a little bit of help, the natural language constructs created by the mango model can be profitably reviewed by the user. The figure shows a prototype of how mango will be deployed as an enhancement to the debugger perspective. The source code view will be as before, but now the breakpoints will control the symbolic execution required to generate the mango model. Instead of threads and call-chains in the debugger view, the peer mango view will describe hierarchical flow-control subdivision into loops and parallel cases. Assumption and Conjecture views will be peers to the familiar variables view, which now takes on the role of describing output state. And of course it will all be hyper-linked

Its going to take some time to complete this port, but I'm working on it. Embedding mango in the Eclipse Workbench will make the tool much more usable and more accessible to a much larger audience. I would be happy to come back next year to EclipseCon and report on the completed tool.

Mango is an open-source sub-project of the JPF project of NASA/AMES. JPF is a core technology at the heart of the mango engine. Here are all the links to the mango repository, the wiki, and the plugin update site. Thank you very much for allowing me to speak today.