

jpf-mango

**NEW!** [Second Example?](#)

[First Example?](#)

**Purpose:** To provide a case by case specification of Java™ source code, analogous to Javadoc but more rigorous. To provide "proof artifacts" to a theorem prover, enabling the mathematical proof of code properties.

**Ultimate Goal:** To integrate the generation of code, specification, and correctness proofs so that a programmer may produce a more reliable product with the same level of effort.

**Theory of Operation:** The code is first converted to a large graph of vertices (instructions) and directed edges (branch conditions), often referred to as the flow control diagram. This conversion occurs at the byte code level, which is convenient because the byte codes are described very succinctly in terms of the state of the Java Virtual Machine (JVM). This conversion is essentially accomplished by the JPF core engine. The Mango formal peer code then generates for each byte code a description in terms of the Mango formal model. A graph subdivision algorithm is applied to the control flow, generating a hierarchical sequence of graphs required to describe the loops within the code as recursive functions. JPF is then used to walk these graphs for the purpose of generating the specification and proof artifacts. The backtracking ability of JPF is utilized to generate cases, and the ability to trace back along trails is leveraged in order to compute loop invariants. Ideally, the process would be fully automated, but in practice when Mango requires guidance from the user, JPF will block and a gui thread will interact with the user to obtain the information required to proceed. The specification and proof artifacts are stored in persistent form exposed to the user in a rule base format, enabling reuse and incremental, distributed operation.

**Limitations:** Mango only specifies "good" cases, exposing input constraints required to satisfy such cases. Some user guidance is required to determine what constitutes a good case. It may happen that even the number of good cases grows exponentially. The user must then provide case generalization logic to abstract away explosive case growth. Such abstractions must also be accompanied by type and translation logic, which the user must provide. Although Mango does generate loop termination conjectures, such conjectures are generally in terms of loop output. Typically the user must provide guidance to form hypotheses for loop termination in terms of loop inputs. Correctness of such hypotheses may be confirmed by an automated theorem prover, which typically requires expert guidance.

**Status:** Mango is based on technology released by the Nasa/Ames Software Release Authority in September, 2008. Much of the original code base was written in C++. By Spring, 2009, the code base was migrated to 100% pure Java, and integrated with Eclipse RCP, the rich client platform. The original code base did not use JPF. Full integration with JPF commenced in the Summer of 2009 and should be complete by January 2010. The tool should achieve an initial operational capability by Spring 2010. Ultimately, the tool may be deployed as an Eclipse workbench plugin.

**Research:** Previous versions of the tool did generate artifacts for the ACL2 theorem prover and substantial proofs were accomplished. Efforts are just getting started to lightly embed the current Mango model in ACL2. However, the proof artifacts are essentially just expressions in the Mango model of definitions, hypotheses and conjectures, and as such are theorem prover neutral. There is no automatic facility for generation of hypotheses on loop inputs required to prove termination. Needless to say, this is a fundamental tool weakness and contributions in this area would be most welcome. (Caveat emptor: there is no general purpose algorithm for determining loop termination, but constructive solutions exist for typical circumstances.)