To support analysis of the system, a custom parser produces a representation of the subject Java program as an abstract syntax tree, and the implementation of the real-world types and type rules produces a database that specifies all of the details of the types and type rules. The abstract syntax tree and the specification for the real-world type system are processed by an analyzer shown in the center of the figure that supports four types of analysis:

- Real-world type checking.

  A *type checker* was implemented for this analysis. It loads the real-world types and type rules, examines each node, especially infix expressions, in the abstract syntax tree, and then checks for violations of real-world type rules. Diagnostics are displayed for users to review.

- Reasonable range analysis.

  A *range analyzer* was implemented to conduct reasonable range analysis. It reads the reasonable range values specified in real-world types and then conducts interval analysis on the Java program. Warning messages are issued when calculated intervals of program elements exceed their reasonable ranges.

- Assertion generation.

  An *assertion generator* was implemented to synthesize assertions as Java fragments that can be inserted into the subject program. These assertions can be used to implement runtime checking of real-world invariants that cannot be checked statically.

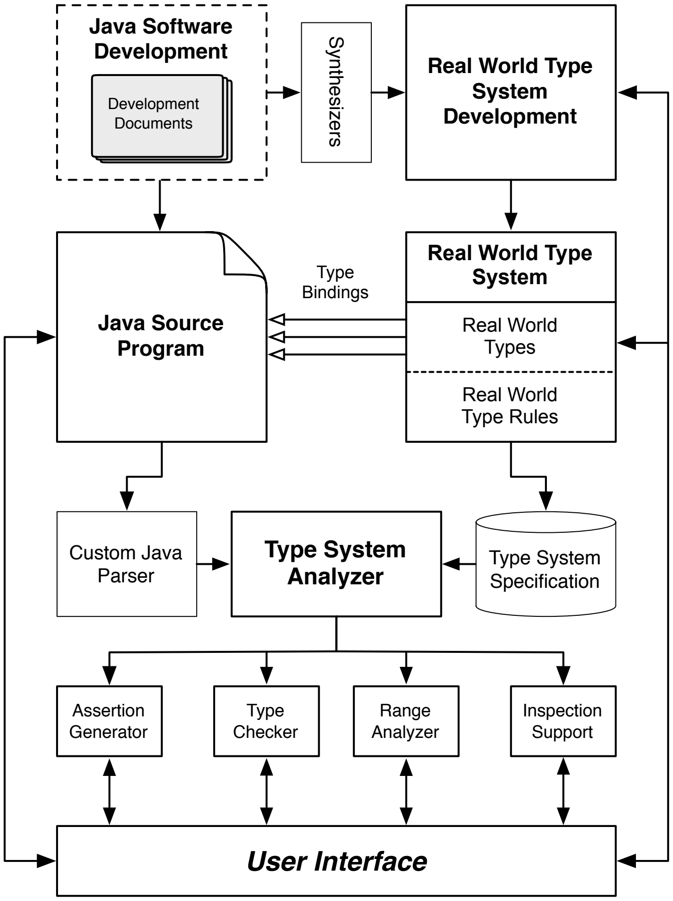

Fig. 2. Architecture of the tool

- Targeted inspection.

  The *inspection support* was implemented to assist human inspections. It has a display called *inspection mode* that reads and displays real-world types for every program element selected by users. It also synthesizes a checklist of locations in the subject program which inconsistent use of real-world types is referred to.

As indicated by the top part, Java programs are separately developed and parsed without being affected by the development of real-world type systems. In this way, the two artifacts can be developed in parallel without impeding each other. Engineers can manually create real-world type systems through the user interface. In addition, the tool implements two features that facilitate developing interpreted formalisms:

- Synthesis of real-world type systems.

  Three synthesizers were implemented for producing candidate real-world types, real-world type rules, and real-world type bindings. The details about the synthesizers were introduced in our prior work [11]. In summary, the synthesizer for real-world types leverages natural language processing techniques to process the identifiers in the program to produce a list of candidate real-world types. The synthesizer for type rules extracts operations that bound with real-world types to produce candidate real-world type rules. The synthesizer for type