Designing a Fully Featured IDE for Engineering Modular, Verified Software

Daniel Welch
School of Computing
Clemson University, Clemson, South Carolina 29631
dtwelch@clemson.edu

Abstract—Integrated Development Environments (IDEs) have made large strides in recent years towards making code easier to write, navigate though, and ultimately maintain in projects of all sizes—large to small. Through features such as syntactic and semantic error annotations, context-based keyword and reference completion, and a host of other transformational “housekeeping” capabilities (such as refactor/rename support), IDEs have significantly reduced much of the burden in arriving at, and ultimately maintaining, syntactically and semantically correct code. This progress report explores the integration of these, and other formal verification related features into a commercial class IDE for RESOLVE—an integrated programming and mathematical specification language.

I. INTRODUCTION

The usefulness and central role of IDEs in developing and maintaining software projects of all sizes is—at this point—undisputed. However, beyond simply helping users write conventionally designed code (checked via IDE-inspections and variety of static analysis techniques), in order to curb complexity while still increasing quality of resulting software, current IDEs stand to benefit from taking up support for languages that allow users to abstractly model components, verify the correctness of implementations with respect to specifications, and ultimately make any such verified code available in easily accessible component-packages for reuse in future projects.

In expanding the role of IDEs to encompass this goal, the question then naturally becomes how to effectively incorporate the RESOLVE modeling, programming, and verification methodology into an IDE in a way that leverages the many positive qualities of existing, modern IDEs while also meshing well with the iterative approach [1] we have found both students and average users to take when reasoning about, and ultimately verifying the correctness of their code. The rest of the paper, which takes several initial steps towards exploring this question, is organized as follows: Section 2 provides some details on the design and high level organization of the IDE, Section 3 highlights some of the current features already integrated, while Section 4 wraps up discussion and details ongoing work—specifically with regards to integrating the vc-generator and prover.

II. IDE DESIGN AND ORGANIZATION

The IDE itself is built on top of the IntelliJ IDEA platform\(^1\) and uses an internal PSI (Program Structure Interface) tree to represent the contents of a RESOLVE file as a hierarchy of elements representing the various constructs in the language. The majority of analysis and features later discussed in this report interface directly with this structure—which allows one to perform fairly sophisticated internal transformations and lookups behind the scenes based on dynamically entered text.

The IDE itself follows a typical organization: there is a main editor window where users spend a bulk of their time writing specifications and implementations, along with a project explorer pane which enables users to browse through the various files belonging to the currently opened RESOLVE project (in addition to the libraries which come standard with the compiler).

III. CURRENT FEATURES

As is typical of IDEs, there are many disparate units of functionality blended together within the editor interface discussed in section 2. In the proceeding sections we highlight some of these features, illustrating their usefulness in writing RESOLVE specifications, theories, and executable code.

A. “Live Template” support

Live templates is an extensible mechanism that allow users to predefine frequently occurring fragments of code, and insert them into appropriate contexts in their programs. A template itself can be thought of as a “document with holes” which users, at the time of instantiation, are prompted to fill in. The actual act of inserting a template into a valid context is performed by typing a label/keystroke (determined by the author of the template) then repeatedly visiting and filling in any blank fields (holes) defined. In this section we illustrate two particular applications of live templates leveraged by the IDE.

1) Mathematical glyphs (Unicode support): A long standing design objective in RESOLVE is in making sure specifications and their underlying mathematics actually look like math—with the expectation that this will make the language more attractive and usable to by-trade mathematicians. This effort however continues to be hampered by the extremely limited range of ASCII characters. While moving to Unicode is the obvious alternative, the biggest barrier by far to its incorporation into any system remains the complicated, unintuitive

\(^1\)https://www.jetbrains.com/
key combinations one must memorize in order to insert such symbols.2

In the interim, until a more universally accepted method for insertion is decided upon, we leverage the power of live templates to make the definition and insertion of such operators in strictly mathematical contexts (e.g. math expression contexts) a feasible option for users. Indeed, having a diverse mix of operators available and (more) easily usable will hopefully allow us to avoid further overloading of common operators (such as $\times$, $\div$, etc) in RESOLVE's mathematical space.

The left portion of Figure 2 shows a panel provided by the IDE that lets users define any operator template along with a label for inserting it.$^3$

2) Language constructs: Besides usage of these templates solely for mathematical symbols, we also take advantage of them as a means of easily inserting larger, sometimes wordy language constructs that are often prone to being mistyped. The listing below illustrates one example of a live template for operations with a procedure body.

Operation $\textit{name}$(params) $\textit{type}$;
    Procedure
        $\textit{end}$
    end $\textit{name}$;

The fields within the $ signs are the holes that are expected to be filled by users at the time of instantiation. Like the Unicode mathematical operators, the list of available templates for RESOLVE constructs is user extensible, but comes packaged with a number standard ‘defaults’ for:

- all module types (facility, precis, concept, implementation, etc)
- variable, facility, operation, and procedure declarations
- type declarations for records, arrays, cart_prods, etc

2For example, the code for the Unicode concatenation ‘circle’ (○) operator is: $\text{alt+0176}$...

3The abbreviation—which one must begin to type to receive a completion prompt—functions similar to systems such as $\LaTeX$ where a set of labels, such as $\lambda$ and $\in$, prompt the insertion of the actual glyphs (in this case $\lambda$ and $\in$, respectively)

B. Keyword & reference completion

As expected of modern tools, the RESOLVE IDE informs users when typing (via a contextual popup box) of possible completions for references to various named declarations (variables, types, operations, etc) as well as language keywords. Figure 2 illustrates an example of keyword completion popup prompting users for a parameter-mode, while the lower two subfigures illustrate examples of reference completion for mathematical specifications.

1) Challenge: keeping specification and implementation completions separate: One notable challenge in implementing reference completion in a language like RESOLVE, which includes both a specification and programming language, is in ensuring that completions for the strictly mathematical objects are adequately distinguished from those for programmatic variables—and that completions for the two do not mix in unexpected, nonsensical ways. More than just making sure
that variables and definitions are suggested in the correct contexts, there needs to be some visual means for making it manifestly clear to users when they are referencing something mathematical in nature vs. something programmatic.

Figure 3 illustrates one (in progress!) solution to this involving icons of a specific color and shape based on the kind of object being referenced (type, module, parameter, field, etc) and its relative persuasion (programmatic or mathematical). These icons would appear next to each completion option as users type.

**Fig. 3.** Some work-in-progress icons for RESOLVE modules and constructs

C. Integration with the RESOLVE compiler: error and warning annotations

The IDE interfaces directly with the RESOLVE compiler, which in turn provides annotations containing token/document position information, along with messages indicating any issues or warnings detected in the code as users type. All semantic checks are performed strictly by the compiler, rather than being duplicated on the aforementioned IntelliJ-specific internal PSI structure. Figure 4 illustrates several simple warnings and errors.

**Fig. 4.** An illustration of several errors and warnings issued by the IDE

IV. RELATED WORK

There are a number of existing IDEs and tools that provide support for verification-related languages. Though for space reasons, we limit discussion to two somewhat closely related tools.

Sireum Logika⁴ is a pedagogical tool designed to help students reason about simple programs in an introductory logic course for computer science students at Kansas State University. The Logika language itself, a so-called “programming logic language”, is marketed as one large language partitioned into subsets: “the propositional logic language is a subset of the predicate logic language, which in turn, is a subset of the programming logic language” ... “which [is ultimately] a subset of Scala”. Like the tool discussed in this report, this language is integrated into the IntelliJ IDEA framework, and uses Microsoft’s Z3 to automatically prove algebraic formulæ arising from student programs—dynamically reporting verification results through the IDE’s interface.

Another tool, this one from Microsofts end of the verification effort, incorporates support for Dafny [3]—an imperative programming language with formal specifications—into an IDE [2] built on top of Visual Studio. Several notable features include continuous processing (proving as users type), dynamic error annotations, “hover over” informational popups, and built-in verification debugging (via the Boogie-Verification-Debugger [4]).

V. CONCLUSIONS AND ONGOING WORK

For the remainder of the summer, I aim to integrate the vc-generator and prover into the IDE. Once this is done, and raw verification information is flowing freely between the two systems, experimentation will hopefully reveal an array of interesting methods for presenting users with VC's and prover results. At least for the initial pass through, I will attempt to replicate fairly closely the way RESOLVE’s existing web IDE handles the presentation of VC's and proof results.

Beyond incorporation of the prover and vc-generator, I hope to continue improving existing features, and eventually exercise the resulting IDE on several of the more complicated component examples, such as a spiral based (heap) implementation of the Prioritizer concept.

ACKNOWLEDGMENT

This research is funded in part by NSF grants CCF-1161916 and DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES


⁴http://logika.sireum.org/