AFRL-RI-RS-TR-2022-165

DEFERRED CONCRETIZATION ADAPTIVE SOFTWARE ENVIRONMENT (DCASE)

PERSPECTA LABS, INC..

DECEMBER 2022

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

Copyright 2021 Perspecta Labs, Inc

THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE US GOVERNMENT.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by AFRL Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2022-165 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM E. MCKEEVER
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED START DATE	3. DATES COVERED END DATE
DECEMBER 2022	FINAL TECHNICAL REPORT	JUNE 2020	AUGUST 2021

4. TITLE AND SUBTITLE
DEFERRED CONCRETIZATION ADAPTIVE SOFTWARE ENVIRONMENT (DCASE)

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-20-C-0519	N/A	62303E
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
		R2Z8

6. AUTHOR(S)
Euthimios Panagos and Simon Tsang

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Perspecta Labs, Inc.
150 Mount Airy Road
Basking Ridge, NJ 07920

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-450

DARPA
675 North Randolph St
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-RI-RS-TR-2022-165

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA#: AFRL-2022-5816
Date Cleared: 6 December 2022

13. SUPPLEMENTARY NOTES

14. ABSTRACT
DCASE (Deferred Concretization Adaptive Software Environment) automates the software sustainment loop using a novel combination of techniques with proven scalability and program-generation speed, while embracing software engineering concepts that ensure ease of use by traditional developers. The DCASE Intent Specification Language (ISL) combines abstraction mechanisms in existing programming language with composition patterns, enabling users to express application requirements and intentions at a domain-specific abstract level, avoiding concretization. Program Pieces, which correspond to reusable, adaptable units of functionality in a target domain, are the building blocks used by a novel combination of Genetic Programming and Symbolic Solving for identifying candidate application solutions and offering specific and targeted information to human-on-the-loop when software generation cannot satisfy updated application requirements. DCASE functionality can be leveraged using any modern IDE that supports the Language Server Protocol, minimizing the learning curve associated with new technologies.

15. SUBJECT TERMS
Intent-Defined Adaptive Software (IDAS), DCASE, Deferred Concretization, IDE, Intent Specification Language, Genetic Programming, Symbolic Solving, Program Pieces.

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	C. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
U	U	U	SAR	63

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
WILLIAM E. MCKEEVER	**N/A**

PREVIOUS EDITION IS OBSOLETE.

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

1.0 SUMMARY 1
2.0 INTRODUCTION 2
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES 5
3.1 INTENT SPECIFICATION LANGUAGE 5
3.1.1 Background: Proposal ISL 5
3.1.2 DCASE ISL Parser 6
3.1.3 TatSu vs Parsimonious 7
3.1.4 IDE Integration 8
3.2 DCASE IDE 9
3.2.1 IDE Development 11
3.2.2 Language Server Protocol Support 12
3.2.3 Using TextMate Bundles 13
3.2.4 IDE Integration with DCASE Storage and Code Generation 13
3.2.5 Solution Viewer 14
3.3 PROGRAM PIECES 17
3.3.1 Design Pattern Program Pieces 20
3.4 GENETIC PROGRAMMING (GP) 21
3.4.1 DCASE Genetic Programing Approach (DCASE-GP) 22
3.5 SYMBOLIC SOLVING 26
3.5.1 Symbolic Solving and Design Pattern Pieces 27
3.5.2 Benchmarks 28
3.5.3 Implementation 29
3.6 DCASE STORAGE 29
4.0 RESULTS AND DISCUSSION 32
4.1 DCASE MVP 32
4.2 IDE EVOLUTION 36
4.2.1 First Iteration: Text Editor Features 37
4.2.2 Second Iteration: DCASE Features Window 40
4.2.3 Running DCASE Generated Code 43
4.3 PERFORMANT EMBEDDED SYSTEMS 44
4.3.1 Elevator System 44
4.3.2 OpenUxAS 45
4.4 END-TO-END INTEGRATION 52
5.0 CONCLUSIONS 54
6.0 REFERENCES 55
7.0 LIST OF ACRONYMS 56

LIST OF FIGURES

Figure 1: DCASE High-level Architecture ... 2
Figure 2: Storage Service ISL ... 5
Figure 3: Simple ISL validation example ... 8
Figure 4: IDE ISL validation button ... 8
Figure 5: ISL parsing error message display in DCASE IDE ... 9
Figure 6: Notional DCASE IDE for application intent specification ... 10
Figure 7: Notional generated code and assurance evidence ... 10
Figure 8: IDE contextual feedback ... 11
Figure 9: IDE Language Server Protocol support ... 12
Figure 10: IDE displaying a sample ISL file using TextMate ... 13
Figure 11: DCASE IDE, Language Server, and back-end component architecture ... 14
Figure 12: Exemplary solution image generated by genetic programming components ... 15
Figure 13: Build status of the backend as seen by the end user ... 16
Figure 14: Solution Viewer mockup showing a solution tree ... 17
Figure 15: GP and Symbolic Solver pipeline for program piece based program synthesis. ... 23
Figure 16: Mapping requires/ensures constraints to Z3 expressions ... 26
Figure 17: Handling program piece cross-reference ... 27
Figure 18: DCASE Web-based storage access ... 30
Figure 19: DCASE application specification results ... 31
Figure 20: DCASE application specification registration ... 31
Figure 21: Chat application architecture ... 32
Figure 22: Chat client and server ISL specifications and developed program pieces (names) ... 33
Figure 23: Order delivery ISL specification and relevant program piece specifications ... 35
Figure 24: DCASE ISL file example ... 37
Figure 25: Auto-complete for new program piece template ... 38
Figure 26: Auto-complete with existing and template program pieces ... 39
Figure 27: DCASE Features Window pinned to the top of the user screen ... 40
Figure 28: DCASE Features Window ... 41
Figure 29: DCASE program pieces have embedded meta-data for tracking provenance ... 41
Figure 30: DCASE Features Window can invoke the default browser ... 42
Figure 31: Features kept from the initial iteration ... 43
Figure 32: Generated application example ... 44

Approved For Public Release; Distribution Is Unlimited.

Figure 33: DCASE solution to an elevator system application 45
Figure 34: GP-based discovery of program piece connections........ 47
Figure 35: DCASE end-to-end component integration 53

LIST OF TABLES

Table 1: Notional DCASE ISL Grammar 6
Table 2: Generic Programming Solution Viewer – Feature List 14
Table 3: Example program piece specifications 19
Table 4: DCASE domain terms and properties for the logistics domain 34
Table 5: Sample OpenUxAS program piece specifications 46

Copyright 2021 Perspecta Labs, Inc

ACKNOWLEDGEMENTS

This research is sponsored by the Air Force Research Laboratory (AFRL) and DARPA under Award Number FA8750-20-C-0519. This report was prepared as an account of work sponsored by United States Government. Neither the United States Government nor any agency thereof, nor any of their employees makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the US Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

1.0 SUMMARY

DCASE (Deferred Concretization Adaptive Software Environment) automates the software sustainment loop using a novel combination of techniques with proven scalability and program-generation speed, while embracing software engineering concepts that ensure ease of use by traditional developers.

The Intent Specification Language (ISL), a key element of DCASE, enables a developer to express the problem to be solved. The programmer uses the ISL to create an intent specification that describes the program's structure in terms of reusable, domain-specific abstractions, and the program's semantics in terms of (1) relationships among abstraction elements, (2) constraints on computational resources, execution environment, performance, and security, and (3) (optionally) symbolic test cases that express the program's expected behavior at an abstract level. Developers use the ISL inside an Integrated Development Environment (IDE) that embodies the ISL and a catalog of **program pieces**. Program pieces are reusable modules that include: (1) an abstract specification describing (in ISL) the structure and semantics of the piece, (2) a concrete part with implementation(s) in one or more target languages (e.g., Java or Python), and (3) assurance evidence for each implementation (e.g., validated via testing or with a formal proof).

DCASE automates program generation (the how) based on the intent specification (the what) by synergistically combining two program synthesis techniques: **Genetic Programming** and **Symbolic Solving**. Genetic Programming explores the problem search space identified by the intent specification and combines existing program pieces to generate candidate solutions (i.e., abstract, ISL programs) that are structurally correct but which may contain pieces with unbound parameters and may not satisfy all intent constraints.

Symbolic Solving assesses each candidate's functional (i.e., semantic) fitness by constructing and solving the conjunction of (1) constraints from the intent specification and (2) constraints implied by the candidate's constituent pieces. If there is no assignment to parameters that can satisfy all the intent constraints, Symbolic Solving provides to Genetic Programming a measure of the candidate's functional fitness, thus helping to bias the evolution of future generations of candidates toward variations that are more likely to satisfy the intent specification. When Symbolic Solving finds a solution, Code Generation uses the candidate's constituent program pieces to emit code in a target programming language. In addition, Evidence Generation produces an assurance argument comprised of evidence for the solution's constituent program pieces and their correct composition (i.e., via pre-/post conditions), the selected parameters, and a representation of the intent constraint satisfied by the solution.

2.0 INTRODUCTION

DCASE (Deferred Concretization Adaptive Software Environment) automates the software sustainment loop using a novel combination of techniques with proven scalability and program-generation speed, while embracing software engineering concepts that ensure ease of use by traditional developers. The high-level architecture of DCASE is shown in Figure 1.

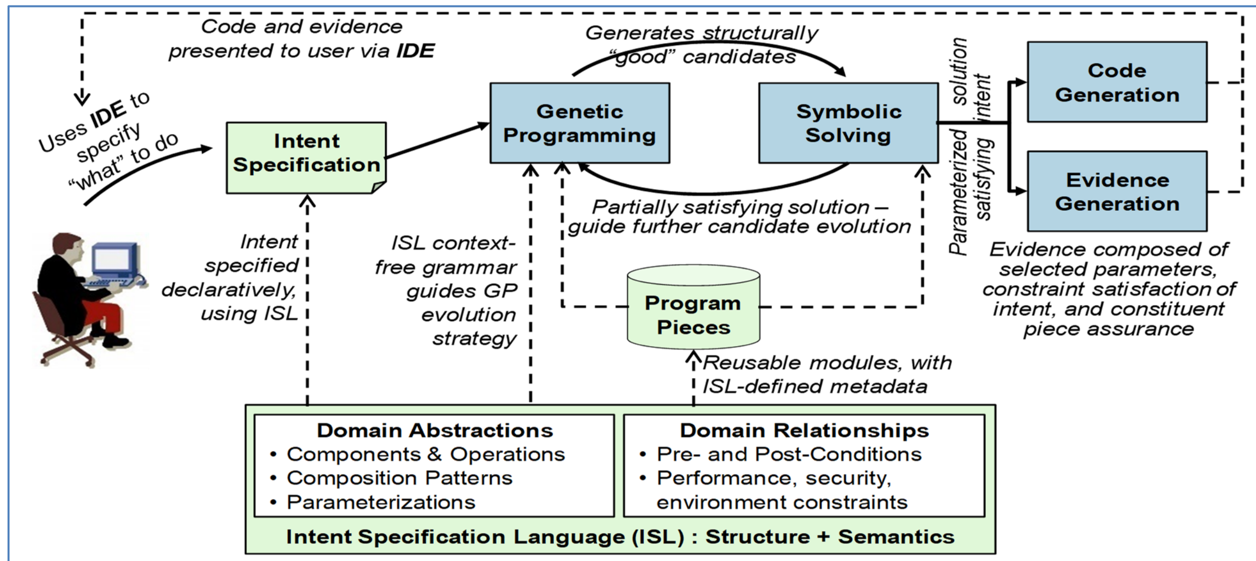


Figure 1: DCASE High-level Architecture

To capture programmer intent while avoiding concretization, the DCASE *Intent Specification Language* (ISL) provides a domain-specific (e.g., logistics or cloud agility) abstract language that enables developers to specify intentions and constraints while deferring concretization to the automated program generation phase. The ISL combines several well-studied abstraction mechanisms, including:

- *Component-oriented programming* [HC01], where a component provides a set of interfaces that collectively provide a useful service (e.g., a data management service with operations to initialize, store, and retrieve data);
- *Generic programming* [GJ+07], to abstract away concrete types and enable the same algorithm to work on multiple types (e.g., C++ templates, Java generics);
- *Configurable software design* [WL96], to parameterize the parts of a program that are expected to change or need to be configured for each deployment environment;
- *Composition design patterns* [GV+94] [HW+04] [BO16], which describe ways to compose higher-level abstractions from primitive ones (e.g., decoupled producer-consumer interaction without concretizing whether it uses a push or pull model); and
- *Assertions* [CR06] or *constraints*, to specify requirements about relationships of abstraction elements that must hold in any correct solution.

The DCASE abstract language for specifying intentions and constraints will be familiar to and adoptable by typical developers because the ISL constructs of components, generics, parameterization, and assertions are contained in mainstream programming languages such as Java and Python. In addition, composition design patterns are popular with developers, as they represent

Approved For Public Release; Distribution Is Unlimited.

a reusable approach to solving a general problem. The DCASE IDE complements the simple and intuitive ISL by providing (1) both *graphical* and *textual* modes for specifying intent, (2) early *syntactic validation* of the intent against the ISL, and (3) *contextual feedback* on inconsistencies or ambiguities identified by DCASE program generation to assist in debugging and updating intent specifications.

The novel combination of *Genetic Programming* (GP) and *Symbolic Solving* automates program adaptation in response to changes in requirements or environment. In addition to automatically generating code from intent specifications by synthesizing a correct assembly of program pieces, DCASE provides invaluable assistance to the developer in creating correct and complete intent specifications corresponding to requirement changes. When program generation fails to find a solution, DCASE provides precise and targeted information to enable the developer to make appropriate changes to the intent specification. Such DCASE human-on-the-loop assistance identifies:

- *Unsatisfiable constraints*, helping the developer to discover and resolve inconsistencies in the intent or between intent requirements and program piece capabilities;
- M*issing implementations* for some abstraction needed for the generated solution, guiding the developer to implement, or find existing code that implements, the piece's abstract behavior; and
- M*issing program pieces* indicating a gap in the assembled program, providing the developer with a description of the behavior of the missing piece (e.g., pre- and post-conditions) that must be specified and implemented.

To generate evidence that revised code correctly implements the change in requirements, DCASE uses a *correct-by-construction* argument based on our hybrid synthesis approach. The assurance argument for intent specification I and generated program P assembled from program pieces $pp_{1..n}$ includes: (1) supplied evidence for each program piece pp_i; (2) GP evidence of structural correctness of the program assembled from program pieces; and (3) Symbolic Solving evidence that the generated program satisfies all constraints. The assurance argument for a change to intent specification I' is similar but based on differences between the original and changed specifications, difference in program piece structure of the original and changed programs, and differences in solved formulas including selected parameters.

If the developer provides *symbolic test cases* as part of the intent, DCASE will produce stronger assurance evidence, as Symbolic Solving further asserts that the generated program (symbolically) satisfies the test cases. Perspecta Labs has successfully used such an approach in the network configuration domain to dramatically reduce the time to design or change a network by generating an emulated network that can be tested (analogous to symbolic execution of developer-provided tests) to verify correctness of the intent specification.

The key to ensuring DCASE scalability to real-world problems is the novel combination of Genetic Programming and Symbolic Solving, using GP to quickly search over the candidate space to identify candidates that are structurally sound, and Symbolic Solving to evaluate their functional fitness. Feedback from the unsuccessful candidates informs GP search adaptation, yielding efficiency. GP is a non-deterministic, parallel, anytime, and adaptive search method inspired by the principles of evolution. It iteratively evolves an initial population of candidates from one generation to the next. In each round, it probabilistically *selects* candidates to become parents for the next generation based on candidate *fitness* and other desirable properties (e.g., novelty), and then subjects the parents to *variation* based on one or more operators (e.g., mutation, crossover), thus producing the next generation of candidates.

DCASE Symbolic Solving uses either a Satisfiability Modulo Theory (SMT) solver or counter-example-guided inductive synthesis (CEGIS) [SR+05] to select appropriate values for a candidate's parameters, resulting in a complete solution that satisfies the intent specification. *DCASE's application of SMT solving is efficient* because the formula provided to any invocation of the solver is based on a *single* candidate generated by GP. This is in stark contrast to approaches that formulate the entire synthesis process as a constraint satisfaction problem, which are intractable for problems of real-world scale. Another factor in reducing the search space is the high-level domain abstractions in the program pieces that DCASE program generation assembles to implement the intended change. DCASE program generation can be easily parallelized, further reducing time to check structural or functional fitness.

Approved For Public Release; Distribution Is Unlimited.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Intent Specification Language

A user comes to DCASE with a set of application requirements, which do not necessarily dictate system architecture or structure. Yet, one cannot avoid mapping such requirements to some form of components that, when combined, constitute a system architecture that satisfies the requirements. The Intent Specification Language, or ISL, is a custom grammar used to capture a DCASE user's requirement constraints. The ISL allows developers to map application requirements to specifications indicating both the software elements and interaction among them needed to satisfy the requirements.

3.1.1 Background: Proposal ISL

In our DCASE proposal, we created a mock ISL file to capture user requirements for a "Storage Service", shown in Figure 2.

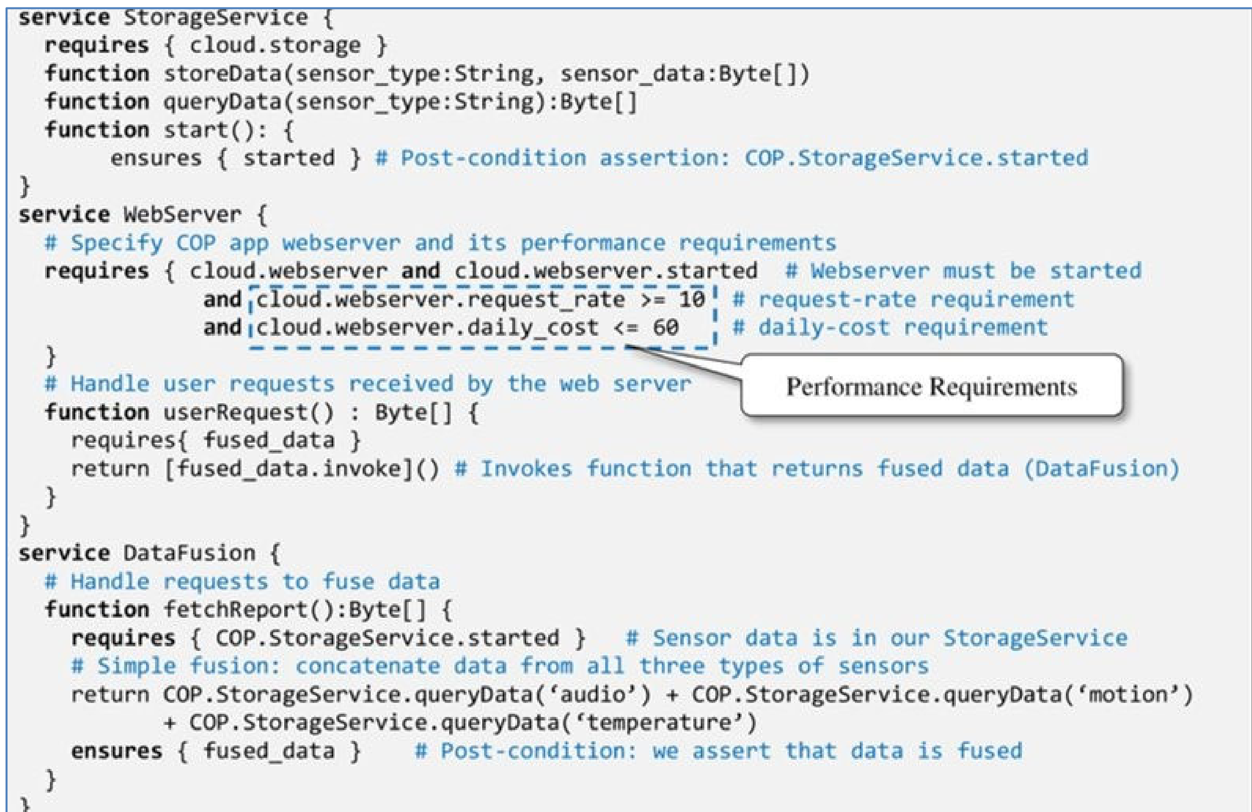


Figure 2: Storage Service ISL

Looking at the specification for just "WebServer", we see the user constraints given in the form of a `requires` clause. It is still human readable, and we can see that the user needed to ensure there was a cloud webserver, that the web server has been started, the webserver could handle at least 10 requests a day, and the daily cost of operation does not exceed $60. The language being used was specified in the proposal as an EBNF grammar, shown below.

Table 1: Notional DCASE ISL Grammar

```
; Application signature -services and assertions.
<application> ::= application <name> { <services> <assertions> }
<services>    ::= <service> | <service> ; <services>
; Service signature -functions, statements, and assertions.
<service>     ::= service <name> { <functions> <statements> <assertions> }
<functions>   ::= <function> | <function> ; <functions>
; Function signature -input parameters, statement, assertions, and return. When referencing
; a function signature in another service, the [ <name> . invoke ] syntax should be used
<function>    ::= function <func_call> : <return> { <func_body> }
<func_call>   ::= <name> ( <parameters> ) | [ <name> . invoke ] ( <parameters> )
<func_body>   ::= <statements> <assertions>
<return>      ::=  /* nothing */ | <type-decl>
; Zero or more parameters can be specified, each having a name and a type
<parameters>  ::= /* nothing */| <parameters> , <parameter> | <parameter>
<parameter>   ::= <name> : <type-decl>
; Statements can include declarations, conditional expressions, and loops
<statements>  ::= <statement> | <statement> ; <statements> | <parameter> = <statement>
                | if ( <constraint> ) <if_part> <else_part>
                | for ( <constraint> ) { <cond_loop> }
                | while ( <constraint> ) { <cond_loop> }
; A statement can be a function call, a nested function call, true or false
<statement>   ::= <func_call> | <dotted_name> . <func_call> | true | false
<dotted_name> ::= <name> | <dotted_name> . <name>
; Conditionals and loops are composed of statements
<if_part>     ::= { <statements> }
<else_part>   ::= /* nothing */ | { <statements> }
<cond_loop>   ::= /* nothing */ | <statements>
; Type declarations for list, dictionaries, arrays, primitive types, and user defined types
<type_decl>   ::= list ( <type_decl> ) | map ( <type>, <type_decl> ) | <type>[] | <type>
<assertions>  ::= <requires> <ensures>
<requires>    ::= /* nothing */ | requires { <cons_list> }
<ensures>     ::= /* nothing */ | ensures { <cons_list> }
<cons_list>   ::= <constraint> | <constraint> ; <cons_list>
<constraint>  ::= forall ( <constraint> ) | exists ( <constraint> )
                | includes ( <constraint) ) | <bool_or>
<bool_or>     ::= <bool_and> | <bool_and> or <bool_or>
<bool_and>    ::= <comparison> | <comparison> and <bool_and>
<comparison>  ::= <expression> | <expression> <compare_op> <expression>
<expression>  ::= <expr_term> | <expr_term> in <name>
<expr_term>   ::= <term> | count ( <term> ) | <term> <arithm_op> <expr_term>
                | <term> <boolean_op> <expr_term>
<compare_op>  ::= == | != | < | <= | > | >=
<arithm_op>   ::= + | - | / | *
<boolean_op>  ::= and | or | not
<term>        ::= <user_term> | <domain_term>
<user_term>   ::= <dotted_name>
<domain_term> ::= <dotted_name>
<type>        ::= String | Integer | Real | Boolean | Byte | None
<name>        ::= [a-zA-Z][0-9a-zA-Z_-]* | ANY
```

3.1.2 DCASE ISL Parser

Traditional parsers, such as Python's `pgen`, were limited by the technology available at the time. Due to memory concerns, most used a single token look ahead, removing contextual data for the parser and decreasing the speed and complexity of the final solution. On the other hand, Parsing Expression Grammars, or PEGs, are unique in that they draw no distinction between lexing and parsing, favoring to do everything at once. This allows many advantages, such as having an infinite

look ahead buffer. Using "packrat parsing," the parser loads the entire text into memory for easy and fast traversal.

PEGs are easier to write (a simple variation of an EBNF) and provide more flexibility in the expansion of the language. With advancements in parsers over the last decade, there has been a boom of PEG-based Python parsing packages available. The DCASE ISL has been implemented using two different parser, Parsimonious (https://github.com/erikrose/parsimonious) and TatSu (https://github.com/neogeny/TatSu).

3.1.3 TatSu vs Parsimonious

When we started implementing DCASE components, we selected the Parsimonious Python package for parsing ISL specifications. Using Parsimonious, we were able to import the ISL specification shown in Table 1 and leverage its 'Node Visitor' feature to traverse the abstract syntax tree and build solvable Z3 expressions for `requires` and `ensures` constraints. However, after integrating the parser with the IDE (see Section 3.2), the limitations of Parsimonious' error reporting became apparent.

Let us use the following typo, mistyping the word '`and`' as an example:

```
ensures { cloud.webserver ad cloud.webserver.started }
```

Parsimonious will detect a parsing error and return a message such as:

```
parsimonious.exceptions.IncompleteParseError: Rule 'assertions' matched
in its entirety, but it didn't consume all the text. The non-matching
portion of the text begins with 'ensures { cloud.webs' (line 1, column
1).
```

This is not very helpful, as it does not show us where the actual break in the grammar was, nor does it supply too helpful of a hint as to where the user should look. The rule 'assertions' is our top-level grammar rule; the entry point into parsing. The creator has stated online that there will be no extended support coming down the pipeline for advanced error handling. Possible workarounds were thwarted too. A special `VisitationError`, used in the Node Visitor to identify where in the generated abstract syntax tree the parser broke, cannot be invoked if the abstract syntax tree cannot be initially created.

TatSu, on the other hand, has exceptional error reporting. Using the same typo from above, TatSu returns the following parse error.

```
(1:27) expecting one of: != < <= == > >= :
requires {cloud.webserver ad cloud.webserver.started
```

3.1.4 IDE Integration

We integrated ISL parsing and validation with the DCASE IDE. In this section, we will walk through the entire parsing validation process to illustrate the integration. First, the user must type their specification into an ISL file using the IDE. In the following simplified example, the user has entered a single `requires` statement, shown in Figure 3. On every `didChange()` event, which is triggered by any user input to the ISL file, the language server protocol (LSP) records the current specification text and stores it in a global variable.

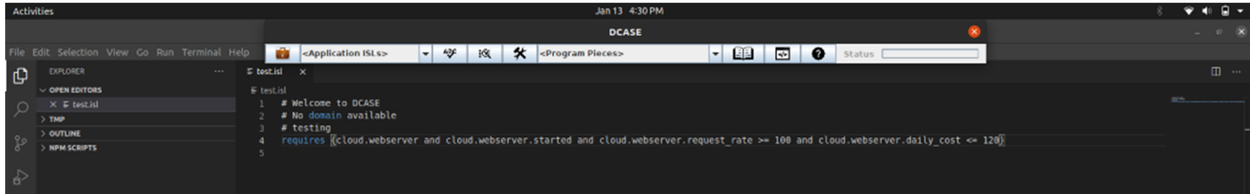


Figure 3: Simple ISL validation example

When the user wants to validate the specification, they click the ‘Spell Check’ button in the DCASE Features Bar, shown in Figure 4. There are currently two buttons to help achieve spell check and validation (the first two buttons from the left after the dropdown), however in its current state the ‘Spell Check’ button (visually defined with an ABC and a checkmark) is the only active button.

Figure 4: IDE ISL validation button

On click, the LSP sends the saved input to the DCASE back-end for validation. Originally, the ISL parser was executed with a system call to the Python parser file. This was changed to favor a system that was not dependent on static file locations. The DCASE back-end receives the text and invokes the generated PEG parser. If there is a parse error, a failed status with an error string describing the error is returned to the IDE. If there was no parsing error, the domain terms and properties referenced in `requires` statements are validated next. If an invalid domain term or property was used, a custom parse error detailing which terms/properties are not supported is returned to the IDE. Parsing errors are displayed by the DCASE IDE using a custom dialog window, shown in Figure 5.

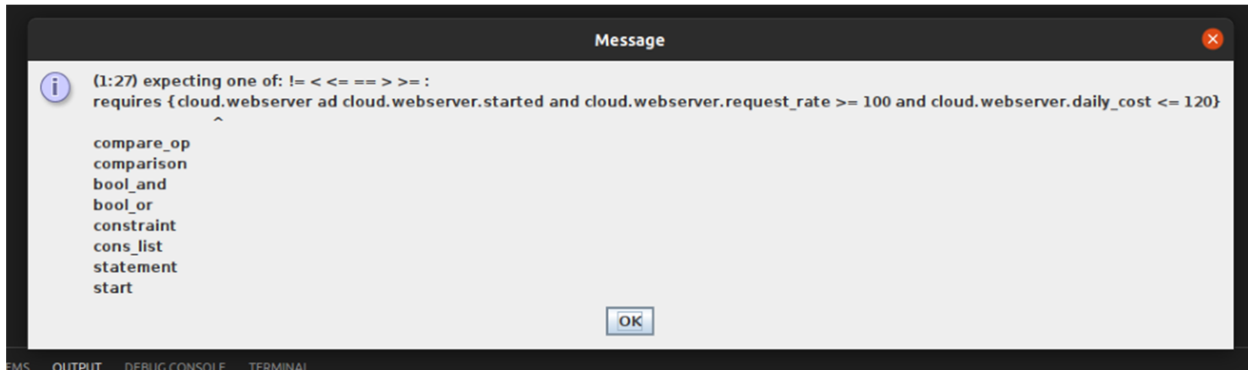


Figure 5: ISL parsing error message display in DCASE IDE

3.2 DCASE IDE

An essential goal for DCASE was to provide the developer or subject matter expert a familiar and intuitive Integrated Development Environment (IDE) with which to specify their intent for a change in requirements, view generated code and evidence, and receive feedback from the DCASE code generation back-end about any errors or ambiguity in the specified intent. In addition, typical IDE functionality, such as autocomplete for ISL keywords, domain semantics, and specifications from the program piece catalog will be available.

In our proposal, we showed a mockup of a notional DCASE IDE as a plugin to Eclipse. Figure 6 depicts the textual and graphical modes of the IDE for a cloud-based application our proposal discussed. When the developer finishes specifying the intent, she presses the "run" icon. The IDE validates that the specified intent is syntactically consistent with the ISL and invokes DCASE program generation. When program generation completes successfully, the developer can view the generated code and assurance evidence in the DCASE IDE, as shown in Figure 7. The IDE checks the new code along with the assurance evidence into the existing project space containing the ISL specification. DCASE provides a "change view" mode to help the developer understand the impact of the change in intent specification. This view compares and displays the difference between the following artifacts:

- Original and modified versions of the generated code;
- Original and modified intent specifications; and
- List of program pieces common to the original and adapted programs, and the ones that are only in the original or adapted versions.

In cases where DCASE is unable to complete code generation due to inconsistencies or incomplete specifications, the IDE provides detailed contextual feedback so that the developer can make the needed changes – for example, to the intent or program piece specification – to enable program generation to complete successfully. Figure 8 depicts the IDE view for the adaptation use case where the intent specification contains conflicting constraints. The developer sees the specific intent constraints and program piece specifications that resulted in DCASE being unable to find a satisfying solution, as well as actions the developer can take to rectify the problem.

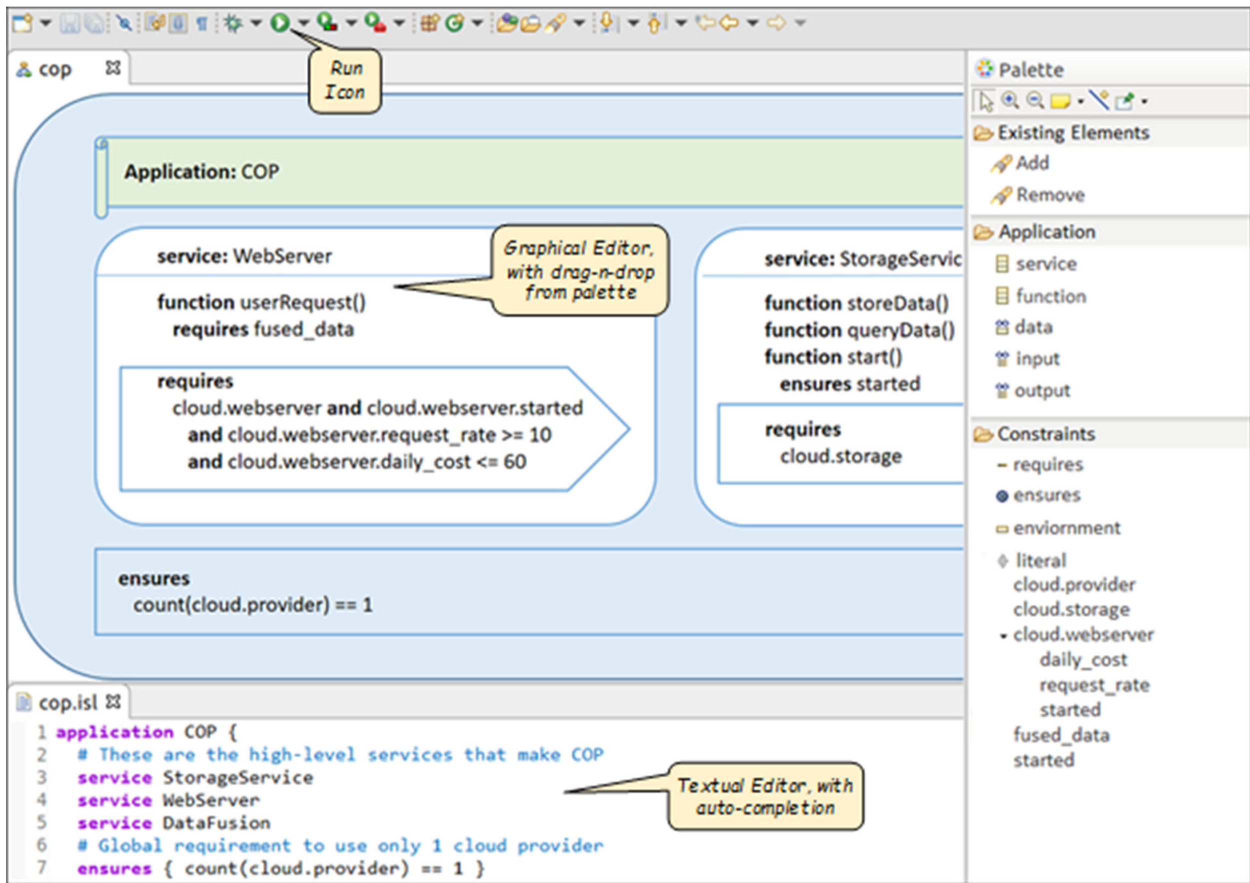


Figure 6: Notional DCASE IDE for application intent specification

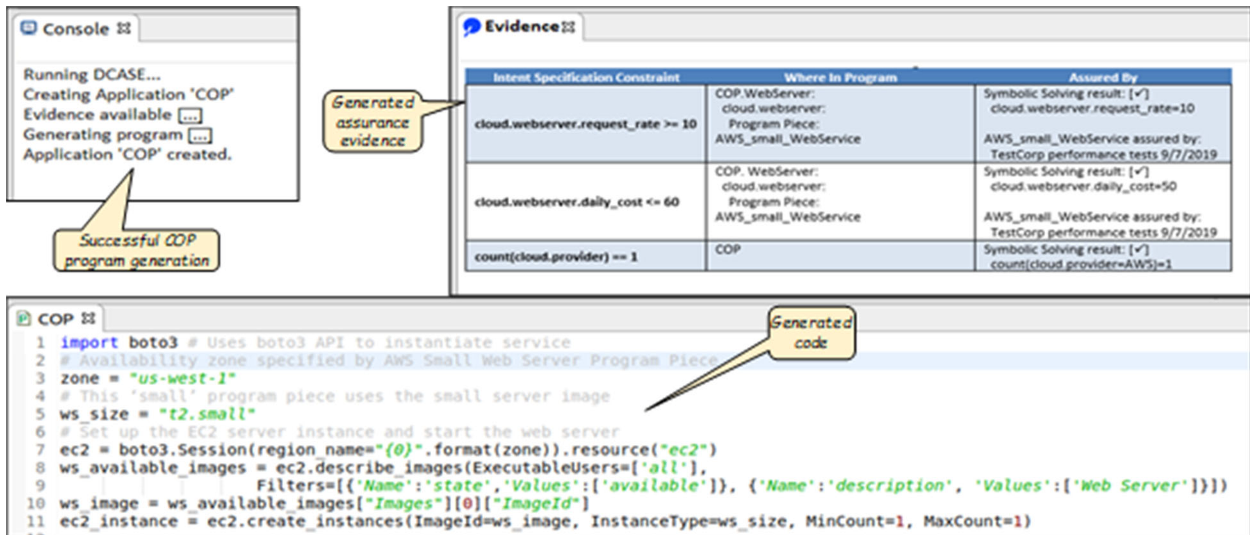


Figure 7: Notional generated code and assurance evidence

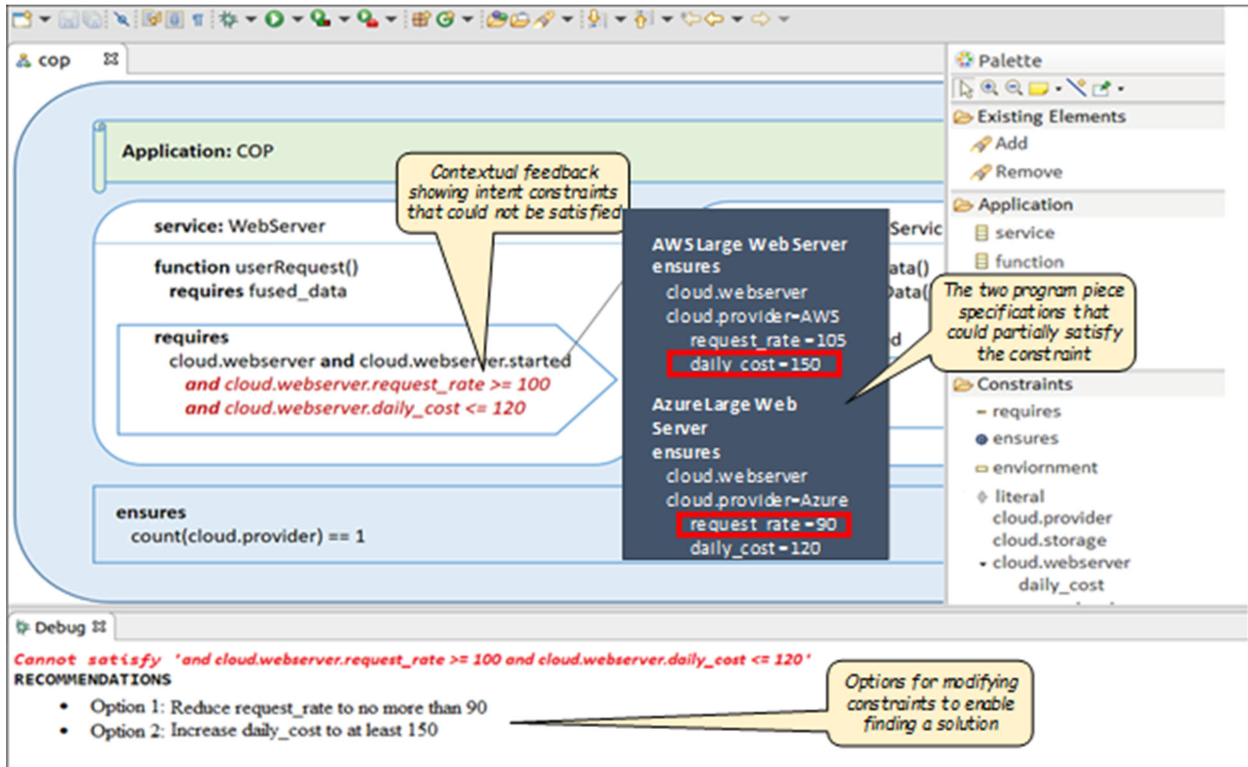


Figure 8: IDE contextual feedback

3.2.1 IDE Development

We had planned to build the DCASE IDE using Eclipse, a widely used software development framework that supported multiple languages as well as code testing and code repository integration. In addition, Eclipse provided a plug-in mechanism that theoretically enabled easy extension to support new languages such as the DCASE ISL. The DCASE IDE would provide both textual and graphical editors for intent specification to meet individual preferences and suitability.

To implement textual aids such as ISL auto-completion and keyword highlighting, we chose the following tools because they were standard across multiple IDEs as well as OS platforms:

- Language Server Protocol (LSP) for providing ISL specific features
- LSP plugin in Java from the Eclipse Foundation
- TextMate bundle for syntax presentation in text editors

Initially, we started creating DCASE IDE using Eclipse with the intention of also porting our results to another IDE such as IntelliJ or Visual Studio Code (VS Code). It turned out Eclipse was cumbersome to use and difficult to integrate even with Eclipse Foundation's own Java LSP plugin. In our initial experimentation, VS Code had a much friendlier user interface, easier LSP plugin integration, and robust IDE documentation. We concluded that VS Code was a superior base from which to build the DCASE IDE.

3.2.2 Language Server Protocol Support

Language Server Protocol (LSP) was an open-source JSON-RPC based protocol that facilitated communication between an IDE and servers that handle language specific features. The protocol was designed to allow IDE independent implementation while providing language specific support, thus allowing the DCASE LSP implementation to be distributed without having to pick target IDEs. Figure 9 illustrates how Eclipse Foundation's lsp4j Java plugin works with Visual Studio Code.

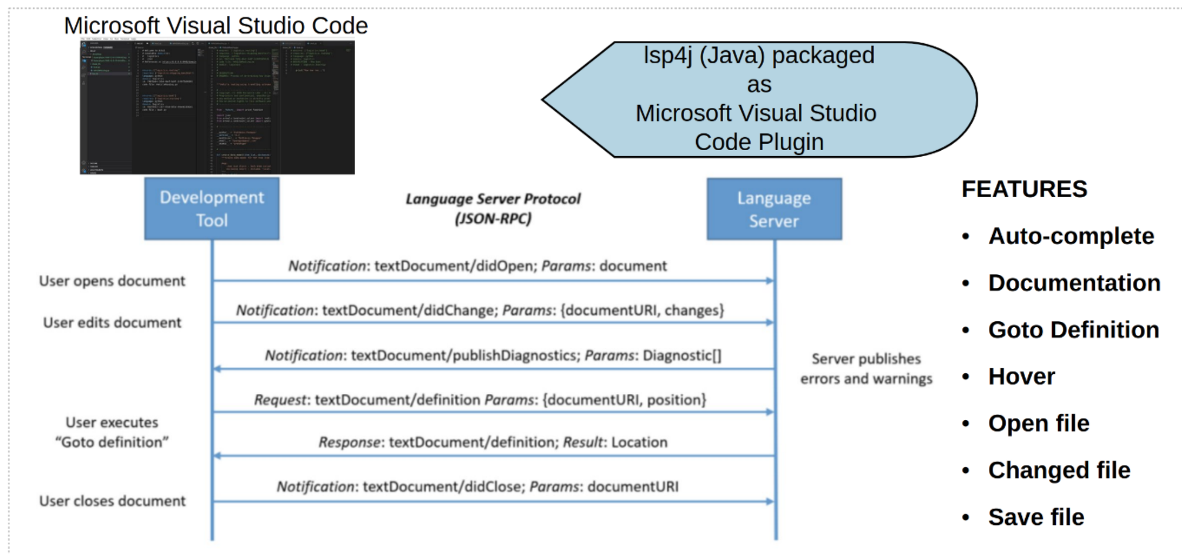


Figure 9: IDE Language Server Protocol support

The main LSP features we planned on implementing were:

- Auto-complete ensures and requires clauses based on chosen domain and available terms and properties in DCASE Storage
- Display documentation and domain term definitions based on available terms and properties
- Hover and/or highlight text to indicate syntax or grammatical issues while user is writes the ISL file
- Hover and/or highlight text to give user feedback after validating ISL but before running either the Genetic Programming Engine or the Symbolic Solver
- Use 'file changed' to trigger syntax and grammar checks
- Use 'file opened' to direct user to a DCASE wiki or tutorial
- Use 'file saved' to trigger a push to DCASE Storage and Genetic Programming and pull the recommended program pieces

3.2.3 Using TextMate Bundles

TextMate bundles were open-source language templates popular IDEs like Visual Studio Code and Eclipse support to customize text editor presentation. Displaying language keywords in different colors is a familiar example of a TextMate bundle application. For DCASE, we were color coding ISL keywords and domain terms via a language grammar configuration file in the VS Code plugin. Figure 10 shows a sample ISL with TextMate color-coding.

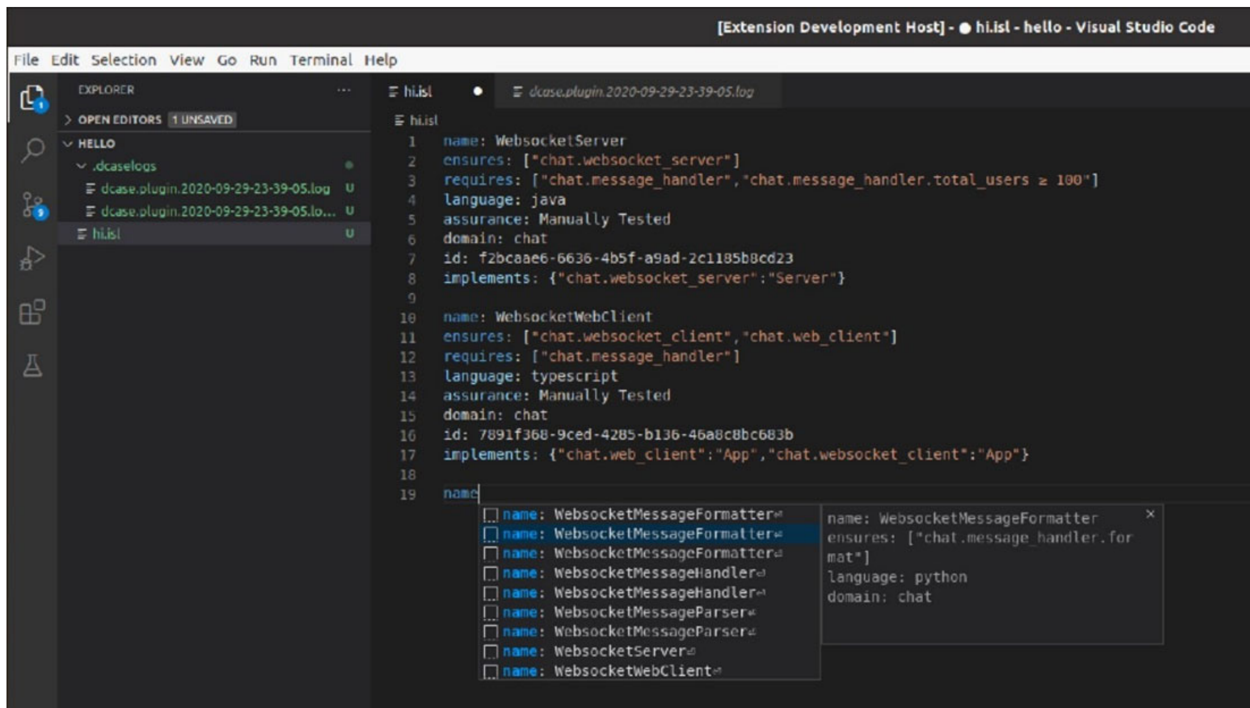


Figure 10: IDE displaying a sample ISL file using TextMate

3.2.4 IDE Integration with DCASE Storage and Code Generation

Once VS Code is up and running with DCASE Language Server and TextMate grammar bundle, DCASE back-end components are accessed via a JSON-based RESTful API, as shown in Figure 11.

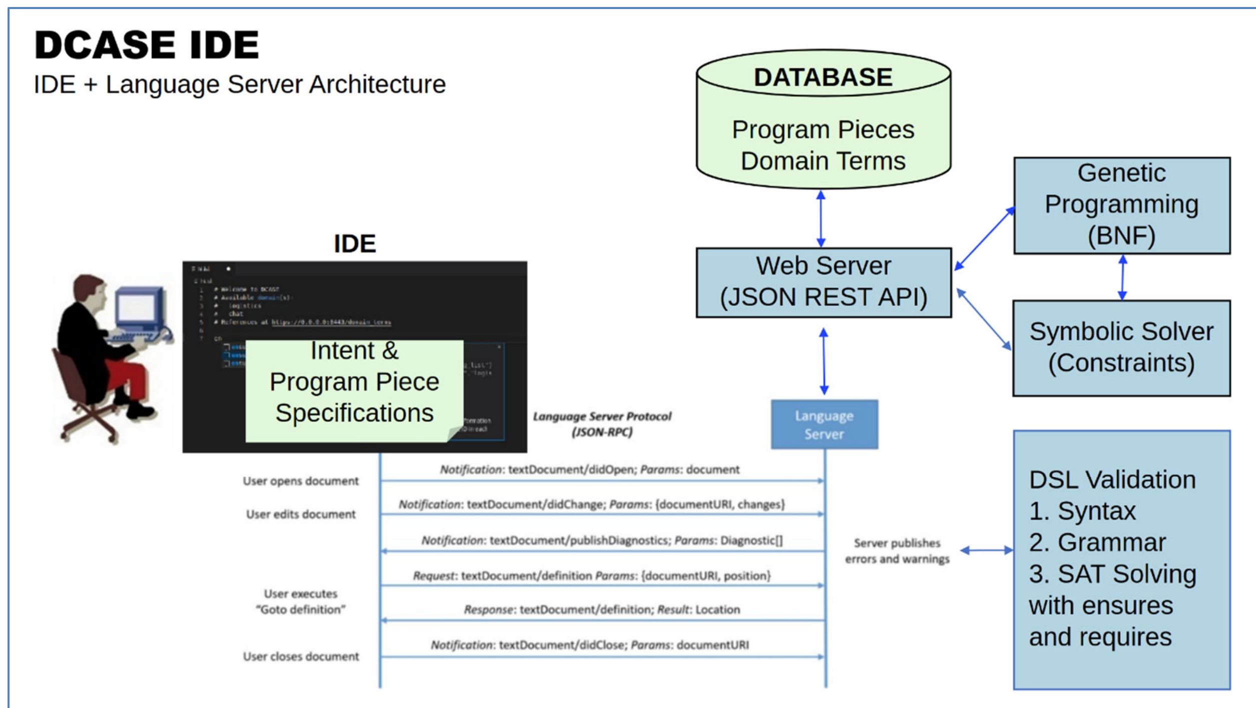


Figure 11: DCASE IDE, Language Server, and back-end component architecture

3.2.5 Solution Viewer

The Solution Viewer (SV) supplements existing capabilities of DCASE and is intended for human-in-the-loop processing use cases. The envisioned features are described in Table 2.

Table 2: Generic Programming Solution Viewer – Feature List

Basic features	• View the status of the Genetic Programming (GP) backend service • View the status of the "build" of a given Application ID • View any solution constructed for this App ID. • Search, highlight, and drill down within the solution. • View supporting "evidence" (if any) provided by the GP component.
Advanced features	• Ability to perform edits and resubmit the solution to the GP component. • Ability to synchronize a solution back to the IDE (Visual Studio Code) for immediate viewing, execution, and more. Syncing to an IDE may be realized by way of one or more intermediate components such as a DB and/or a message bus, etc.

A typical use case involving DCASE SV tool may be as follows:

1. User is assembling some DCASE ISL for an App (appid: XYZ) using the IDE.
2. She invokes the "build" capability of DCASE.
3. User opens the SV viewer and views the status of the build engine for App XYZ. This page updates intermittently and provides a live dashboard for the solution set. The build status may be, "still working", "failed", or "completed". Any given generated solution may have

a status of satisfied or unsatisfied and may or may not contain “partial” or “hypothetical” Program Pieces.
4. User selects a particular solution and the SV generates an interactive view of the solution tree (see Table 2).
5. If satisfied with the solution the User chooses to send it to the IDE for execution and/or further and immediate source code editing.

Notable functional requirements on the implementation of Solution Viewer include the following:

1. Manage and present solutions effectively regardless of their size. E.g., some solutions may be composed of many (hundreds) of Program Pieces.
2. Provide a strategic combination of “big picture” and “detailed” views of the Solution so the user may easily make sense of the solution structure, related back the Requires, Ensures, and Constraints originally designed into the ISL.
3. Provide clear and simple access to evidence.

In addition to the above, or as a temporary measure, the ability to display the big picture solution as an image is viable. The reason for this is that GP components presently have the ability – through Graphviz API’s - to output `.png` images of solutions (see figure below).

Figure 12: Exemplary solution image generated by genetic programming components

These images effectively depict solution trees of nodes; the nodes are interrelated by parent child relationships shown as labeled arcs between them. Note that while these images are static, and large images could pose legibility/usability problems, it is possible that they will nonetheless be useful in some use cases.

3.2.5.1 List of candidate technologies

The following technologies are likely useful in the implementation of the Solution Viewer.

- JavaScript – the building block for front-end capabilities
- jQuery – a JavaScript API with many Document Object Model (DOM) and Ajax related

utilities

- Bootstrap – a responsive Cascading Style Sheet (CSS) framework with useful layout control and visual components
- Fancytree.js – an open library for building tables and tree tables with interactivity in the browser.

3.2.5.2 Solution Viewer Design Mockups

This section illustrates potential design layouts and stylings for the Solution Viewer, and some (but not all) of the features described earlier in the previous sections.

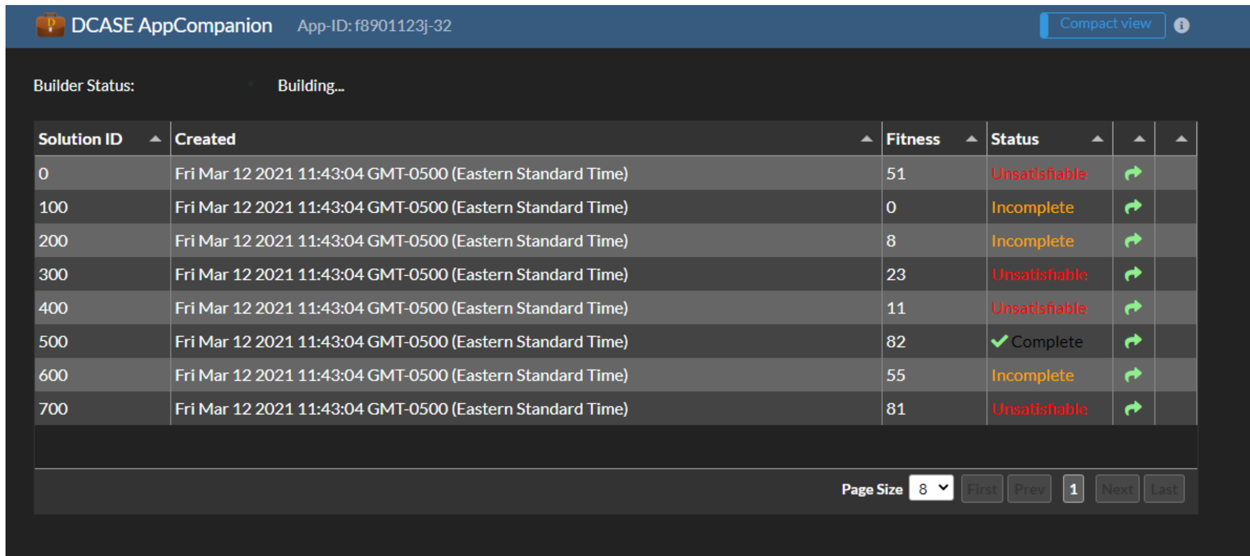


Figure 13: Build status of the backend as seen by the end user

Figure 13 illustrates a mockup for the build status component. In this view, the user is presented with a continually updated view of the solutions built for a given application ID. Every solution has an execution status. Clicking on any solution shows the user a Solution Viewer that allows detailed navigation of the solution along with evidence.

Figure 14 illustrates a mockup for the Solution Viewer. In this figure, the user examines a particular solution using a web browser. Buttons on the top navigation toolbar enable functionality such as:

- Send/sync this solution to IDE
- Resubmit edited version to GP engine
- Toggle edit mode
- Show other formats of this solution

Approved For Public Release; Distribution Is Unlimited.

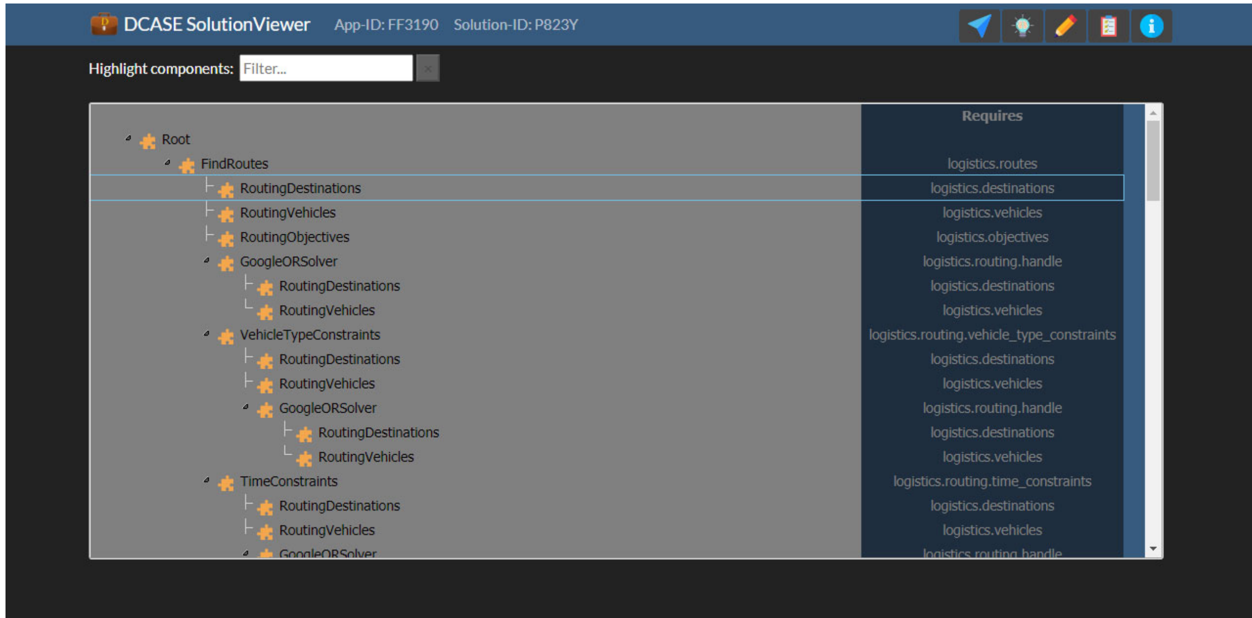


Figure 14: Solution Viewer mockup showing a solution tree

Figure 14 illustrates a search bar (with text label, "Highlight components") for finding, highlighting, and editing particular components. The main panel illustrates a solution set in tree form. Important aspects of each node of the tree are highlighted in a column to the right of the tree. These aspects may include a list of Requires, application constraints, or other information. The implementation should provide various search and navigation capabilities.

3.3 Program Pieces

We chose to write program piece specifications in YAML for two main reasons: it is easy to write and understand, and IDEs have support for mixed language sub-blocks. Program piece implementations can be either stored in the own files or included in the YAML specifications, as shown in

Table 3. In addition, program piece implementations use an existing programming language (e.g., Java, Python, C, C++, JavaScript, etc.). This avoids the need to introduce yet another language for expressing implementations.

Approved For Public Release; Distribution Is Unlimited.

Table 3: Example program piece specifications

Program piece specification: **assignment.yaml**

```yaml
piece: Trailer2TractorAssignment
specification:
  requires:
    - logistics.packing_list
    - logistics.tractor
  ensures:
    - logistics.shipping_manifest
implementations:
  python:
    code: file:assignment.py
```

Program piece implementation: **assignment.py**

```python
import json
import copy
import random

class Assignment:
    """Simple assignment of trailer to tractors."""
    def __init__(self):
        """Default constructor."""
        self._manifest = []

    def assign(self, packing_list, tractor):
        """Assign trailers to tractors.

        Args:
            packing_list (list) : Array of packed trailers
            tractor (list) : Array of tractors

        Return:
            True if assignment was successful. False otherwise
        """
        available = copy.deepcopy(tractor)
        for packed_trailer in packing_list:
            trailer_type = packed_trailer['trailer']['type']
            was_assigned = False
            for atractor in available:
                if trailer_type in atractor['trailers']:
                    trailer_copy = copy.deepcopy(packed_trailer)
                    trailer_copy['tractor'] = copy.deepcopy(atractor)
                    self._manifest.append(trailer_copy)
                    available.remove(atractor)
                    was_assigned = True
                    break
            # If we failed to find a compatible tractor for this
            # trailer, we declare an assignment failure
            if not was_assigned:
                return False

        return True

    def __str__(self):
        """Pretty printing of our state."""
        return json.dumps(self._manifest, indent=4)
```

Approved For Public Release; Distribution Is Unlimited.

```
    def shipping_manifest(self):
        """Return shipping manifest."""
        return self._manifest
```

Combined program piece specification and implementation: **message_handler.yaml**

```
piece: WebsocketMessageHandler
specification:
  requires:
    - chat.message_handler.parse
  ensures:
    - chat.message_handler
    - chat.message_handler.total_users =
                 chat.message_handler.parse.strings_per_second / 5
implementations:
  java:
    imports:
      - import chatServer.messages.ChatMessage;
      - import chatServer.Session;
      - import chatServer.Server;

    code: |
      public void onMessage(Session session, ChatMessage message)
             throws IOException
      {
        for (Server s : others) {
            synchronized (s) {
                try {
                    s.session.getBasicRemote().sendObject(
                           <%= chat.message_handler.parse.expr %>);
                } catch (IOException | EncodeException e) {
                    e.printStackTrace();
                }
            }
        }
      }
    assurance: Unit Test Suite
```

Depending on desired granularity level, program pieces may depend on other pieces. In this case, the notation **<%= … >** is used for representing such dependencies. Program piece size provides a means of trading off solution level with compositional flexibility and solution adaptability. Large (or high-level) program pieces may be specialized, resulting in reduced flexibility and higher adaptation complexity. On the other hand, fine-grained program pieces result in great flexibility and adaptability at the expense of harder Genetic Programming and Symbolic Solving process.

3.3.1 Design Pattern Program Pieces

While using DCASE to prototype our initial chat application, we discovered that in some cases, we needed program pieces that had "holes" in class or method name positions in the pieces. In addition to the use cases in the chat app, another general application for such holes is to support program pieces for "design patterns," which are reusable recipes for solving common programming problems. For example, we might create program pieces for the Observer pattern (also called Publish-and-Subscribe) by having two classes, shown below.

```
class ?Subject {
       void ?attach(?Observer o) { ... }
       void ?detach(?Observer a) { ... }
}

class ?Observer {
       void ?update() { ... }
}
```

Here the names prefixed by "?" are holes, e.g., the name of the `class? Subject` can be solved for, as well as the names of the methods `?attach` and `?detach`. Then if our target app needs to use the Observer pattern, DCASE can instantiate those pieces – possibly multiple times – by choosing appropriate solutions for these names.

So far, we have been exploring this idea in the context of Java, though the same ideas will apply to other languages, such as C and C++, and to other common programming idioms even if they are not usually called "design patterns."

Program pieces carry a type along with them: they can be *definition*, *shred*, or *test* pieces. Definitions contain entire Java class definitions, while shreds contain only certain class members (i.e. fields and methods). Test pieces contain a program that can be executed (i.e. a class with a static `main` method), which will print some information to the console to describe how successful the tests were. Success in these tests is defined by how many assertions passed vs. the total number of assertions encountered.

Program pieces are combined to form a solution to the intent specification using Genetic Programming and Symbolic solving. As part of the combination, the resulting code is analyzed using a custom-built type checker. This type checker defers failure for as long as possible, such that it can suspend its checking in the presence of holes, and resume checking in the presence of a (hypothetical) mapping. The hole-solving stage takes the set of holes needed to solve, together with the constraint graph generated by the type checker, and produces an output mapping from holes to names, or failure, with an indication of "how far" it was able to progress. This, in turn, can be used by the GA to assign a fitness to an individual candidate.

3.4 Genetic Programming (GP)

To date, program synthesis has been approached as a search problem. This sets up two challenges. First, the space of all possible syntactically correct programs is so large as to be intractable to enumerate, so some sort of heuristics or downscaling has to be employed (to the detriment of accuracy). Second, it is hard to verify that the program meets all the design intents because some are either left latent to make formal verification work or they are expressed as input conditions that are too unwieldy to analyze. Search space intractability has been addressed using a variety of techniques, including heuristic enumeration, deduction, constraint solving, genetic programming, machine learning, and neural program synthesis. Capturing user intent has been addressed using formal logic specifications (mostly for deductive techniques), informal natural language descriptions, and templates (e.g., sketches), among others.

In Genetic Programming, the individuals in the population are computer programs. These programs are typically represented as trees, and new programs are created from two parent programs by removing branches from one tree and inserting them into another. This simple process ensures that the new program is also a tree and so is syntactically valid. Other representations can be used as long as they also ensure syntactic validity, e.g. graphs. To guide evolution, Genetic Programming runs test cases on candidate solution. It then uses a fitness function to measure the quality of test case responses (usually program outputs). One challenge of Genetic Programming however is that designing a fitness function for complex domains is non-trivial.

3.4.1 DCASE Genetic Programing Approach (DCASE-GP)

Existing genetic programming program synthesis approaches rely on a grammar for constructing candidate solutions. In many cases, the grammar production rules are augmented with type information for generating syntactically correct candidates. In DCASE, no explicit grammar is specified. Instead, DCASE derives production rules automatically from domain specific terms and properties, composition patterns and requires/ensures constraints. In addition, program synthesis is constructed as a pipeline with three stages: program piece assembly, name solving, and type solving. Symbolic solving and GP can each perform all or none of the stages, or combination to perform them. To evaluate the pipeline design, we compared results of our integration with random search and enumerative search. We performed better than random and were more efficient than enumerative search.

Note that the problem of program piece synthesis that we are tackling is completely new to GP and requires a radically novel setup. In the GP community, program synthesis is usually accomplished by using the test case pass rate as the fitness function, where a test case evaluates a synthesized program on a set of input conditions and compares the synthesized program's outputs to desired ones. This requires test cases and evaluation of test cases but incorporates no notion of formal correctness being checked or used in synthesis. In the GP community, there is truly little research into combining formal methods and GP for program synthesis.

We tried multiple representations of program piece candidates within our extended GP system. The challenge is to find a representation that can be integrated with Symbolic Solving and be suitably varied by GP variation operators to generate interesting and diverse candidate programs that both explore and exploit relative to current solutions. One of them is a directed acyclic graph (DAG) approach that offers advantages over tree-based reorientations in some cases. Tree-based representations enable reuse by subtree replication. This, however, implies that two equivalent subtrees would have to be mutated the same way at the same time, an unlikely case. DAG-based representations, on the other hand, enable more efficient program piece reuse by eliminating the need for simultaneous mutations. Preliminary results based on the chat application mentioned earlier and 100 independent trials indicate that DAG-based representation result in 100% successful solutions compared to only 18% when using tree-based presentations.

3.4.1.1 Genetic Programming and Symbolic Solving Integration

Program pieces represent solutions to simpler or similar problems in the same domain as the program synthesis task. They are components, tagged by constraints, i.e. *requires* and *ensures*, giving them semantic "structure", which allows them to be assembled only in valid ways. As such, they can be seen as multi-use puzzle pieces that, once assembled in one of many permissible ways,

result in a complete solution. With Genetic Programming aiding such assembly, an application intent-specification is progressively decomposed and solved by an aggregation of pieces. When no program pieces are available to satisfy specific requires and ensures constraints in the application specification, the DCASE-GP presents these non-satisfiable requires and ensures constraints and “asks” the developer to implement them or handles the missing pieces (see below). Once Genetic Programming has assembled candidate programs that satisfy the highest level of intent, the result is sent to symbolic solving.

At a high level, we represent the application intent-specification as a program piece that has unfulfilled requirements, positioned as a root of a program tree. The current search starts from this root piece. It expands the piece using stochastic depth-first traversal with backtracking to assemble initial candidate program piece combinations that ensure *requires* constraints of each program piece. The candidate program piece solution is then evaluated by a symbolic solver. Then, in an iterative search loop, the candidate solutions are modified and evaluated again, as illustrated in Figure 15.

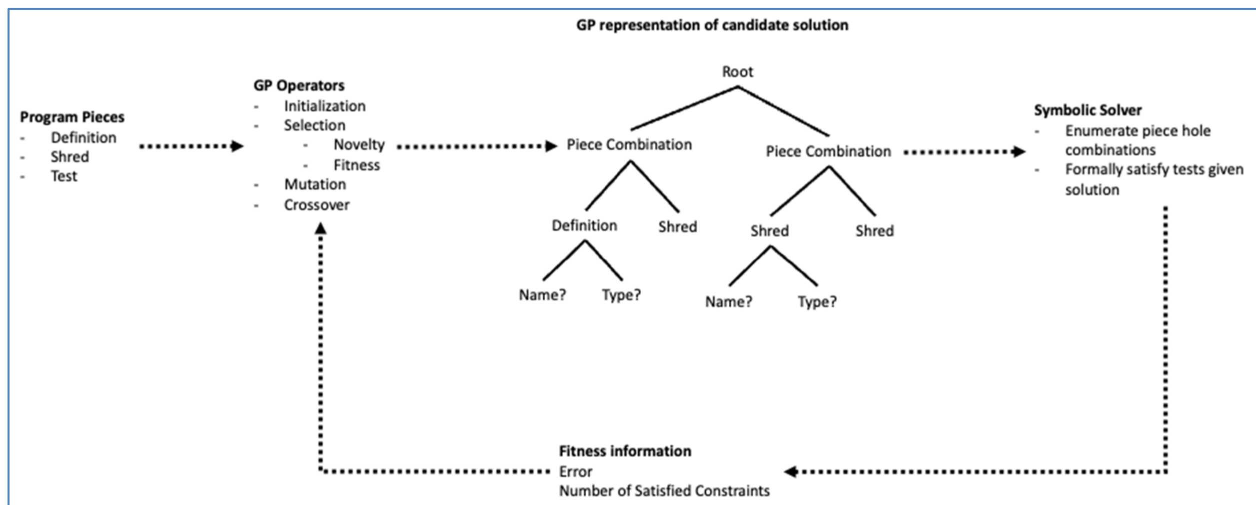


Figure 15: GP and Symbolic Solver pipeline for program piece based program synthesis.

3.4.1.2 Missing Program Pieces

The above approach works fine when all *requires* constraints of program pieces in a solution are satisfied by *ensures* constraints of other program pieces in the same solution. One possibility to handle missing pieces is to use "partial" candidate solutions. A partial solution starts from the root and tries to return the solutions that are closest to a fully ensured candidate solution based on the existing pieces. We implemented this and carried out an initial investigation on how the algorithm can discern which may be better solution (based on the quality of the partial solution).

Another approach we investigated was the creation of "hypothetical pieces". A hypothetical piece is automatically generated by Genetic Programming to ensure the unsatisfied requirements. The goal of this approach is to present the user with a suggestion of a piece that could be created in order to develop a solution based on the intention specification. Currently, after candidate program pieces are initially selected by Genetic Programming, we create hypothetical pieces to reconcile the difference between *requires* and *ensures* constraints of all the available pieces for the domain.

A specific hypothetical piece based on each un-ensured piece is created.

When the search is started and runs, these prepared hypothetical pieces are treated the same as true pieces but some combinations comprising partial solution will reveal new missing pieces. Genetic Programming combines these into a "hypothetical link piece". To do this, it starts bottom-up, rather than root-down and links two pieces that exist with a hypothetical one, thereby connecting the partial solutions.

We have explored both the tree and graph representation for finding solutions to the "missing piece" problem. We have also improved the output format and reporting from the Genetic Programming and Symbolic Solving DCASE components to provide JSON files. We have started with a proof-of-concept for the hypothetical pieces for the logistics domain. We have also continued basic research in the area of Genetic Programming and program synthesis and submitted a paper to a conference regarding called "Getting a Head Start on Program Synthesis with Genetic Programming".

The work tries to incorporate more domain knowledge into Genetic Programming program synthesis. Teachers often give their students a head start. In programming courses, they demonstrate how to solve either a problem related to one coming up on the problem set, or a problem that is possible to re-purpose, by making minor variations. We explore how to give Genetic Programming a head start to synthesize a programming problem. Our method uses a related problem and introduces a schedule that directs it to solve the related problem first either fully or to some extent first, or at the same time. In addition, if the related problem's solutions are written by students or evolved by Genetic Programming, we explore the extent to which initializing the Genetic Programming population with some of these solutions provides a head start. We find that having a population solve one programming problem before working to solve a related programming problem helps largely as the targeted problems and the intermediate problems themselves are selected to be more challenging.

3.4.1.3 Fitness Function

To guide evolution, genetic programming uses a fitness function for evolving the candidate pool. However, designing a fitness function for complex domains is not trivial. In DCASE, we take a different approach by utilizing program pieces as both guiding and building component for synthesizing complex programs. Program pieces correspond to solutions to simple problems in the same domain. As such, they can be seen as puzzle pieces that, once combined in specific ways, they will result in a solution to the more complex problem. When no program pieces are available to satisfy specific requires/ensures constraints in the application specification, DCASE identifies requires/ensures constraints that need to be satisfied by the missing pieces and “asks” the developer to implement them.

Traditional Genetic Programming techniques require input-output examples for evaluating fitness. In DCASE, such input-output examples may not be available. To address this challenge, we utilize Symbolic Solving that explores solutions based on requires and ensures constraints associated with the application specification and candidate program pieces. In addition, we introduce structural measures that also contribute to fitness calculations (e.g., number of program pieces used). The DCASE approach is driven by the definition of program pieces and their special designations of *requires* and *ensures* constraints, and another which is driven by a desire to defer complete

concretization of the synthesized code to as late as possible in the program synthesis workflow, so that Symbolic Solving can be used in tandem with Genetic Programming. Our proposed innovation centers on combining Genetic Programming with program analysis in a way to reap the best of both approaches: efficient search and composition from Genetic Programming and program analysis with Symbolic Solving. To this end, we are utilizing program pieces as both guides and building blocks for synthesizing complex programs.

Our initial implementation of the fitness function computed a fitness value by running the constraint solver over the combination of program pieces in a candidate solution and counting how many constraints the solver was able to satisfy; more constraints solved means higher fitness. If the solver fails, that is the extent of the combination's fitness, since there is no way to compile the program (since there is no type-correct choice of names and types). On the other hand, if the solver succeeds, we include another fitness component by measuring the number of test cases the resulting program passes, if available. In DonkeyGE, the GP tree representation is collapsed into a piece combination list from which a Hole-to-Name map is extracted and passed to the symbolic solver to fill type holes. The symbolic solver returns errors (e.g., errors can be 'Missing Class names`) or some number of satisfied constraints. These are also integrated into fitness for the GP search. We are also investigating additional measures that are returned from SymSolver and how they can be used to assign fitness to a GP individual.

We performed a stand-alone study regarding fitness functions and GP that we call: Synthesizing Programs from Program Pieces using Genetic Programming and Refinement Type Checking. For the study, we implemented a Liquid Haskell version of GP and selected three appropriate benchmark problems. We added GP refinement type checking and integrated type check accuracy along with syntactic correctness into the fitness function. We compared the refinement type checking with test case execution and random search and observed that GP with refinement types finds solution faster than using test cases for fitness. This is an early indicator that refinement type checking could be helpful for guiding the GP to improved program synthesis. We are in the process of writing a submission tentatively for EuroGP 2022.

3.4.1.4 Operators

Ideally GP will use operators that balance its "exploration" and "exploitation" while considering context to avoid being too random. DCASE uses bespoke initialization, mutation and crossover, as well as Novelty Search to help guide the GP search. The initialization operators have tunable probabilities of generating initial GP solutions of different sizes. This is used to focus early on small combinations of pieces. Novelty search guides the GP search by considering tree distances among GP programs as a fitness component, in addition to the performative fitness component. This allows the search to avoid early convergence. For tree mutation, we implemented adding, deleting or replacing existing Piece Combination, Definition or Shreds, as well as replacing existing Hole-to-Names in the map. For crossover, we implemented swapping Hole-to-Name maps by representing them as lists and using one-point crossover.

Here we also performed a separate study called: Using Knowledge of Human-Generated Code to Bias the Search in Program Synthesis with Grammatical Evolution, which we presented at Genetic and Evolutionary Computation Conference (GECCO) 2021 as a poster. A longer version is in preparation for EuroGP 2022. In the study, we use information extracted from a human authored code corpus (GitHub) to guide the search. Various software metrics are incorporated. We observe

that the performance of programs synthesized by GP on test cases does not improve compared to standard GP "test case" fitness. However, the human generated code information improved the "complexity" (e.g. use of control flow operators), readability and maintainability of the programs synthesized by GP. Thus, this study showed how other information could be provided to GP to improve the search performance on different measures.

3.4.1.5 Implementation

We refactored our GP system called DonkeyGE to Java to improve integration with Scala-based symbolic solver. DonkeyGE stores its individuals as in-memory Java trees (recording both piece structure and hole names). It uses the SymSolver API to generate valid symbolic solver input and generate a fitness value for each individual. It can then use either this fitness or its built-in novelty metric to select the best individuals from each generation and create the next generation of candidates. DonkeyGE can choose to fill in all the names, some of the names, or none. Its search performance of remains to be improved.

3.5 Symbolic Solving

Requires/ensures clauses may include properties that Genetic Programming cannot really reason about by itself (e.g., Boolean expressions). Such clauses are passed off to a symbolic solver, which generates a single, large query to judge whether they are correct, or how close to correct they are. In our very first implementation, constraints where mapped to Z3 expressions (parsed using Parsimonious) and their fitness value was set to 1.0 or 0.0, as shown in Figure 16, depending on whether all constraints were satisfied or not.

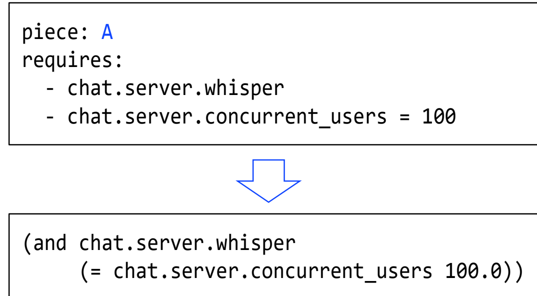


Figure 16: Mapping requires/ensures constraints to Z3 expressions

Once GP and Symbolic Solving find a solution that satisfies all required and ensures constraints of the application specification, application source code is generated. Generated source code includes:

- Source code from program pieces in the DCASE library whose specifications satisfied some of the application requires/ensures constraints
- Source code from new program pieces (application code) written by the developer

The current implementation also supports cross references between program pieces using textual substitution prior to emitting the final code, as shown in Figure 17.

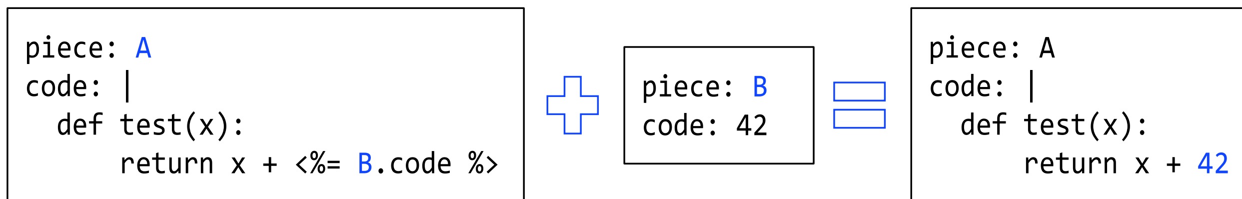


Figure 17: Handling program piece cross-reference

3.5.1 Symbolic Solving and Design Pattern Pieces

We have been exploring design pattern pieces in the context of Java. One key component of the system we are building is a novel solver for constraints among names and types in Java program pieces (this is the "symbolic solver" in the standard DCASE architecture). Constraints in the solver have the form `T1 ≤ T2`, meaning `T1` must be a subtype of `T2`. The key novelty of our solver is that both names and types can be unknowns. For example, we can have a constraint of the form `A ≤ [φ:B]`, meaning that `A` must contain a field of type `B`, but the name of the field is some unknown φ we need to solve for.

Our initial approach was to use a brute-force constraint solver that can exhaustively try all name and type solutions for holes, backtracking when necessary. After implementing an initial version of this approach, we applied the solver to a set of benchmarks (see below) and found that the subset of Java our solver worked on was missing language features we needed for the benchmarks. Thus, we added support to the solver for various looping constructs, operators, primitive types, generics, and arrays, among others. We also added support for using a portion of the Java standard library.

Generics were perhaps the hardest challenge to tackle, as it affected every stage of the solver's pipeline, from how it ingested Java code, to how it stored its constraints, to how it evaluated those constraints during the hole-solving stage. Following that were the primitives; Java has notions of primitive widening and primitive narrowing, and these need to be encoded correctly in order to run the experiments we were writing.

The enhanced solver worked on a complete Java program with holes. However, for our use case, we wanted to combine separate program pieces to form a Java program, and then run the solver. We think of piece combination as an operator ‡, such that if we have pieces `A` and `B`, `A ‡ B` creates a new piece `C` with the members of `A` and `B`. There were many small decisions to answer about what this means, but two of the bigger ones were: first, what happens when two pieces are combined and they share member names? Our solution was to alpha-rename everything in the right piece (`B`, in the case of `A ‡ B`). Similarly, we needed to answer what to with the class name, inheritance, and interface implementations when two definition pieces were combined. We chose a similar answer: prefer the structure from the piece on the left (`A`).

Note that the combiner can also be run on program pieces that are individual methods or fields rather that complete classes, simply by wrapping such methods and fields in a fresh class and then applying the merger. Thus, given a set of pieces, the combiner can merge various subsets of them to form a complete Java program containing one or more classes, which we can then run the constraint solver on.

We abstracted the solver and combiner into a `SymSolver` API, available in both Scala (which our constraint solver is written in) and Java. The API provides utility functions for working with pieces, such as merging sets of pieces together and then using the merged pieces to create a

complete Java program; pre-mapping holes to names, as used by the GP component; and displaying the fitness of assignments.

To support searching for a solution using GP, next we implemented a prototype notion of _fitness_ for a given combination of pieces. The fitness is computed by running the constraint solver and counting how many constraints the solver was able to satisfy; more constraints solved means higher fitness. If the solver fails, that is the extent of the fitness computation, since there is no way to compile the program (since there is no type-correct choice of names and types). On the other hand, if the solver succeeds, we can compute another fitness by measuring the number of test cases the resulting program passes.

3.5.2 Benchmarks

This quarter, we continued working to develop a benchmark suite to evaluate the scalability of our tool on real-world Java source code. We found a popular GitHub repository containing Java implementations of a wide range of standard data structures and algorithms. From this repository, we chose the ten largest programs to form the basis of a benchmark suite.

Using a binary-search tree data structure as a test input, we identified language features that were not accepted by the tool, and either extended the tool to handle them (as discussed above) or developed a simple syntactic transformation to eliminate that feature. Essentially, we aimed to support "interesting" features in the tool and leave "uninteresting" ones to the future. For example, our tool does not support private, protected, and public access modifiers, so we simply remove them.

Next, we explored various ways to turn the benchmarks (which are whole Java programs) into sets of program pieces. Our core approach is to extract a benchmark's methods into individual, separate pieces. We are currently working out the details of how fields and constructors should be handled. This approach gives us a large set of program pieces to run experiments on.

Preliminary experience running on the benchmarks, even with our non-fully-working code, has revealed that at first glance the search space for the solver is enormous, but there are many potential optimizations to greatly reduce it. In particular, a number of places were identified in the hole-solving phase where the system was doing wasted work. It would exhaustively explore search spaces that it could have pruned at a much earlier point in its search. To address this, we implemented heuristics to identify such situation and noticed an order of magnitude improvement in some of our test cases. In addition, adding these heuristics allows the system to reject certain invalid piece combinations wholesale, and return a fitness result detailing this invalidity to the GP.

3.5.3 Implementation

The symbolic solver is written in Scala, and is a collection of loosely coupled modules that can be set up to form a pipeline. In order to facilitate the brute force comparisons with DCASE-GP, as well as to provide a simple interface for the DCASE-GP to use, we developed a shared API that wrapped the modules into simple ordered steps, and then mirrored that API for use with Java, which is what the DCASE-GP is written in.

To report fitness, we realized that we had several discreet states that the system could end it — success, and different kinds of failure. We also realized that these types formed a lattice, and could be used directly by the DCASE-GP as fitness results. An example of these are the aforementioned Invalid Candidate fitness, but there is also a Constraint Failure fitness, that gives the maximal number of constraints that an individual was able to satisfy, as well as the total number of constraints needed for it to pass that stage.

3.6 DCASE Storage

Domain-specific terms, properties, as well as program pieces, application specifications, and generated solutions are stored in a DCASE database. Database entries are accessible via a number of mechanisms, including direct database access (via a wrapper API), RESTful API, or a Web browser. The RESTful API and Web-based Dashboard to support the following functionality:

- Querying domain terms and properties
- Storing program pieces
- Updating program pieces and domain terms (including properties)
- Deleting program pieces and domain terms (including properties)
- Saving and querying application specifications
- Initiating application build process and retrieving build status

The RESTful API is used by both the IDE and Genetic Programming. The IDE interacts with the storage component during auto-complete of domain terms and properties (as part of application specification creation), when saving or retrieving application specifications, when saving or retrieving program pieces, to initiate the application "build" process, and retrieve application status, including generated code.

Figure 18 shows the results of a search query for program pieces. By clicking on the View button, one can view all available implementations for the selected program piece. Figure 19 shows the results of a search query for application specifications.

Figure 20 shows the web-based save form for an application specification. Each application specification must reference an application domain (e.g., cloud, logistics) and be assigned a name. The name does not have to be distinct (a unique ID is assigned by DCASE). An optional version number can also be included. The application specification is expected to be stored in a JSON or YAML encoded file.

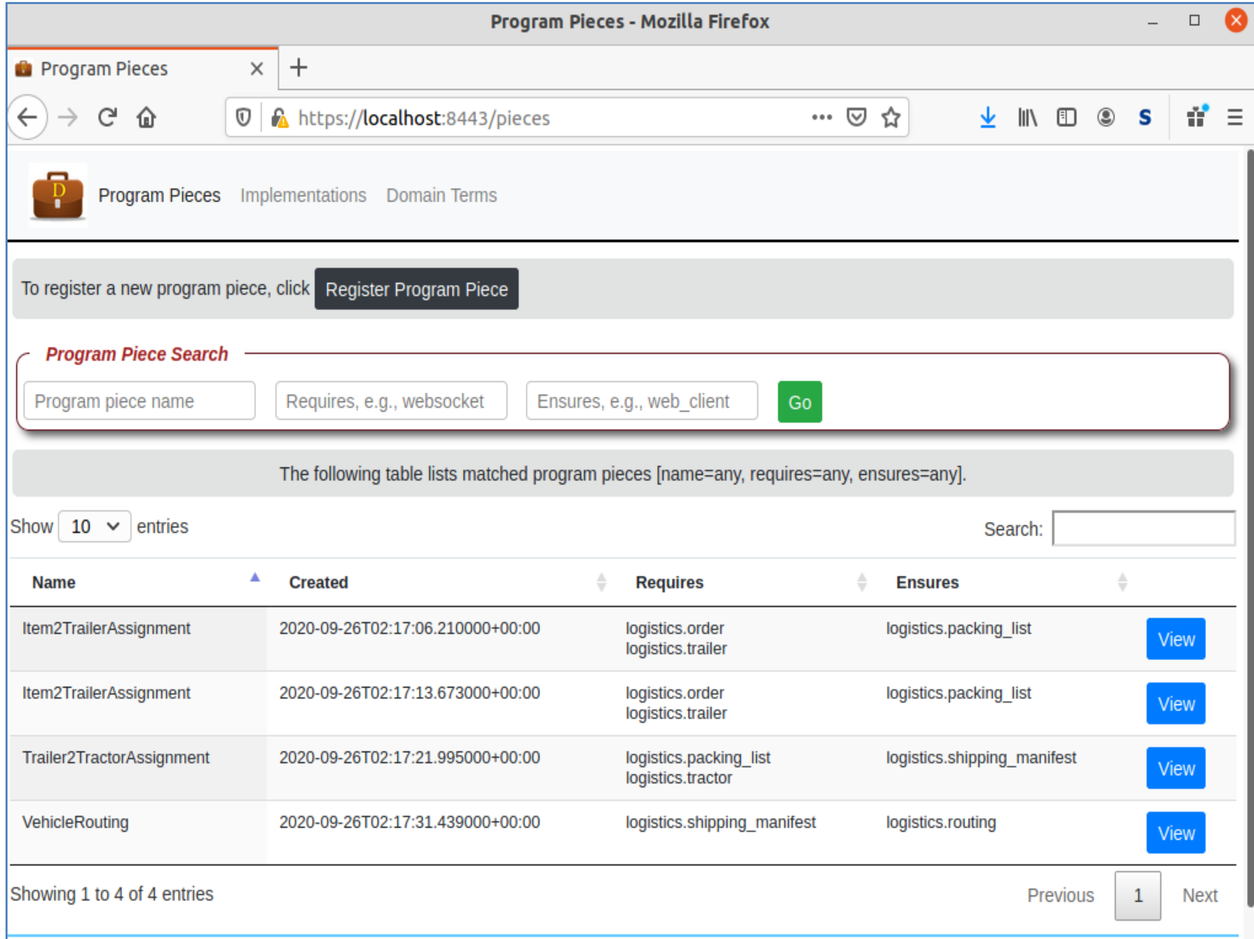


Figure 18: DCASE Web-based storage access

Genetic Programming interacts with the storage component during candidate solution generation. Given a new application specification, Genetic Programming queries the storage component for program pieces that satisfy specific *ensures* and *requires* constraints. As the candidate solution is evolved, additional queries may be issued against the storage component. When a candidate solution is found to satisfy all application requirements, the solution is saved with the storage component.

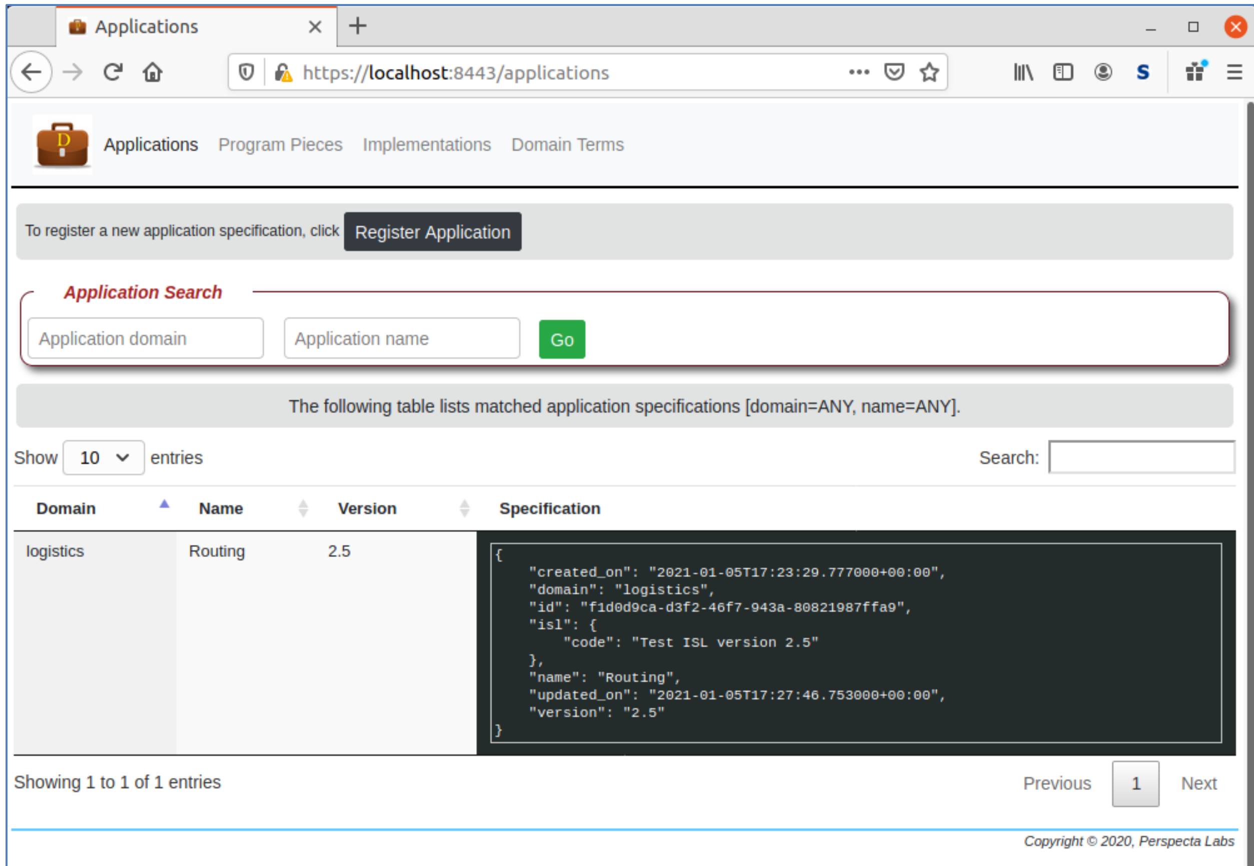


Figure 19: DCASE application specification results

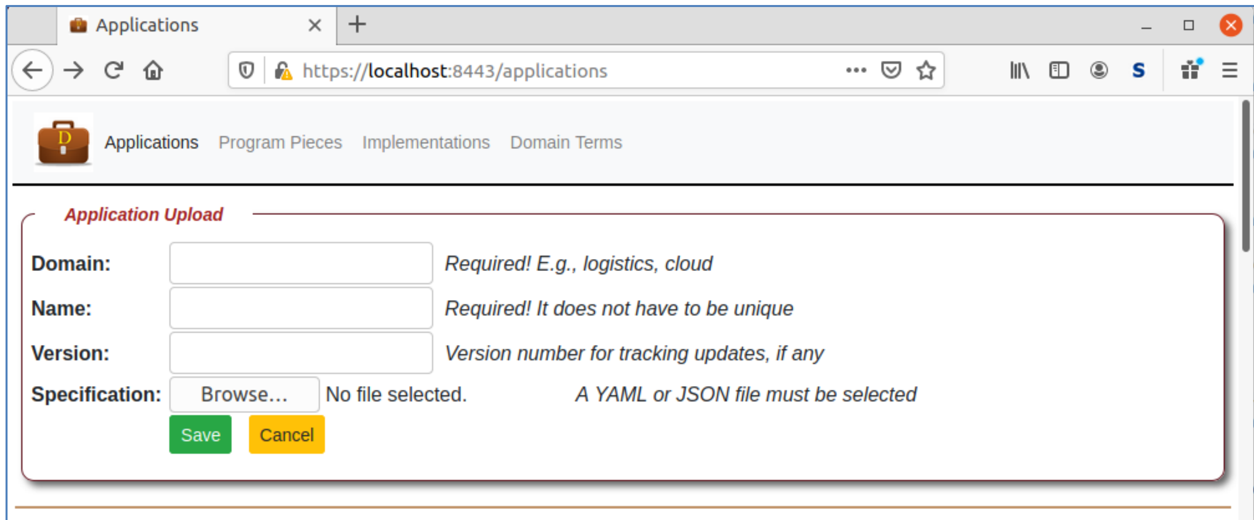


Figure 20: DCASE application specification registration

4.0 RESULTS AND DISCUSSION

Our main goal since the project kickoff meeting was to gain a better understanding of the core research challenges in realizing the DCASE vision. To achieve this, we focused on developing a minimum viable product (MVP) version of DCASE by implementing initial versions of its core components: ISL, program pieces, Genetic Programming, Symbolic Solving, and IDE.

4.1 DCASE MVP

We focused on two simple applications for this purpose: a chat application from the cloud domain, and an order delivery application from the logistics domain. For each application, we created an initial list of high-level requirements. Then, we followed two different paths in order to explore how requirements can be mapped to intent specifications.

The first path, applied to the chat application, followed a traditional early concretization approach where a developer implemented a version of the application based on the requirements. Once the application was developed (using a combination of a Java back-end server and a JavaScript Web front-end), we started breaking up both the back-end and the front-end into **program pieces** and creating their abstract specifications in terms of *requires* and *ensures* constraints. Such constraints are part of the DCASE ISL and are used for encoding application requirements in the target domain (cloud in this case).

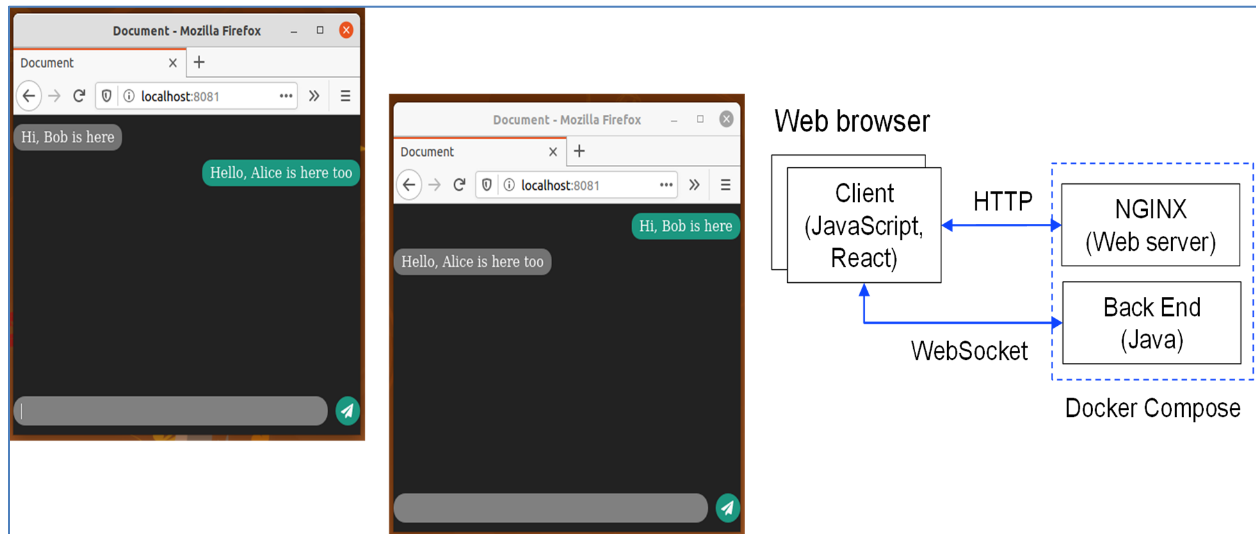


Figure 21: Chat application architecture

After we were done with the initial breakup of the chat application, we expanded the initial set of requirements (just broadcast of messages to all connected clients) to include: (1) support a simple IRC-like command set, (2) allow users to assign names to themselves, (3) send private (whisper) messages to each other, and (4) attach images to their profiles. To be able to express these requirements using the DCASE ISL, we introduced more domain terms and properties to be used in requires and ensures program piece specifications and implemented additional (specialized) program pieces. Figure 21 illustrates the chat application architecture. Figure 22 shows the resulting chat client and server specifications and program pieces (names only) we developed for

satisfying the constraints imposed by these specifications.

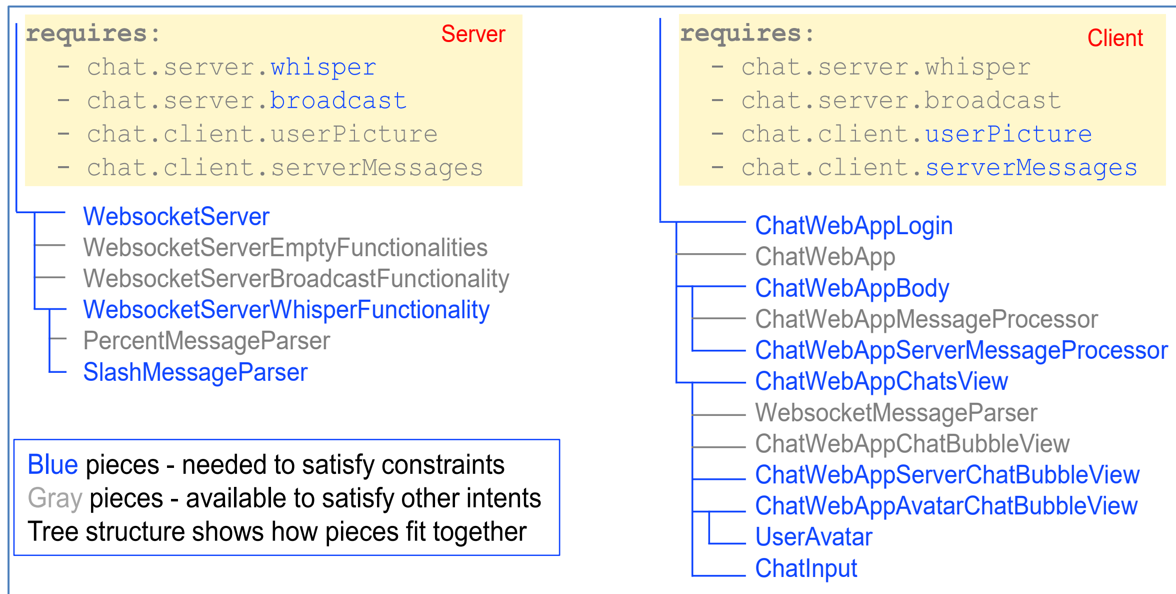


Figure 22: Chat client and server ISL specifications and developed program pieces (names)

The second path, applied to the order delivery application, followed the envisioned DCASE approach, where a developer first created an intent specification for this application by using domain-specific terms and properties. Next, the developer applied her experience in using a divide and conquer approach to create program pieces that could be used for satisfying the application specification. For this application, we used the following initial set of requirements:

- Transport items from a central warehouse to their destinations
- Items are rectangular in shape
- Items cannot be placed on top of each other
- A limited number of rectangular-shaped trailers is available for loading items
- A limited number of tractors is available for transporting the trailers
- Minimize the total distance travelled for delivering all items

To improve our understanding of the logistics domain, we considered all the abstracts in the 2020 International Conference on Computational Logistics, which gave us a broad overview of an interesting set of challenges in this domain. As a result, we identified several interesting adaptions, including adding:

- Minimization of a carbon tax or other supplemental charge to the problem;
- Hard or soft time constraints (e.g., the hours a truck could arrive at a depot);
- Additional constraints regarding warehouses, such as warehouse regulations, capacity, or number of loading docks; and
- Additional constraints regarding drivers, including driver wage, availability, and security clearance.

We then explored sources for logistics program pieces source code. As a starting place, we found that the operations research community has developed many solvers that can be used for such

problems. Many of these solvers have their own specification languages classified under the umbrella term algebraic modeling languages (AML). We also found a Google-developed, open-source framework called Google OR-Tools, which provides an abstraction layer for specifying such constraints, which provides a unified way to use any underlying solvers with ease. Google OR-Tools also provides a high-level framework customized for the vehicle routing problem (VRP), which translates into the underlying constraint problem. Unfortunately, this framework might not be flexible enough to encode all the variations we want to explore. For this reason, we decided to encode our problems using both the vehicle routing framework and the underlying constraint programming language.

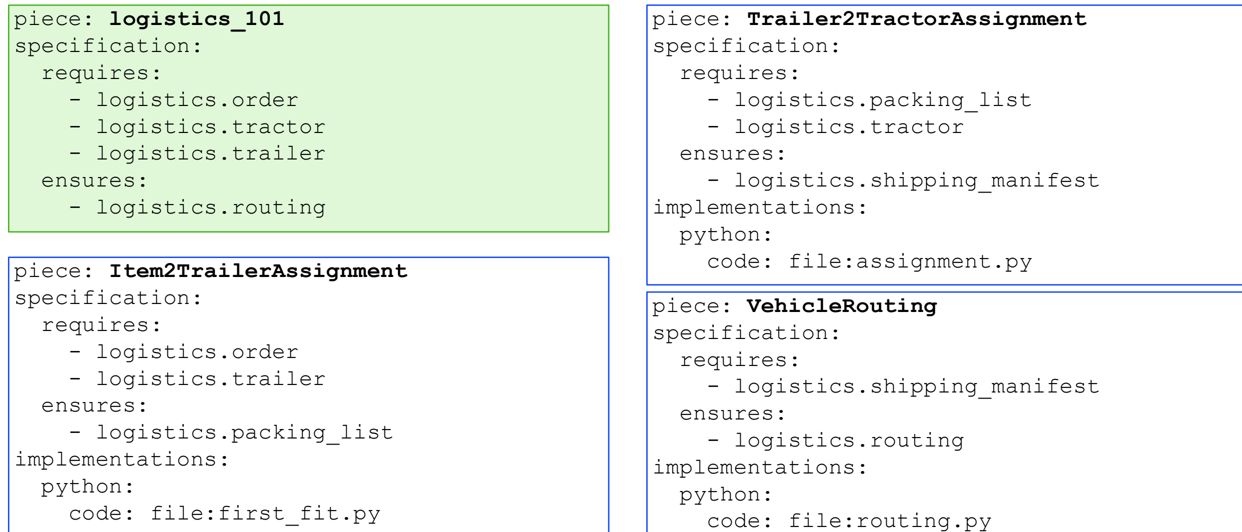
We encoded the basic vehicle routing problem, in which we have a set of vehicles emanating from a single warehouse and transporting goods to multiple locations, both using the VRP framework and as a constraint problem, where the objective is to minimize the distance traveled by the vehicles. We then began the process of designing and implementing program pieces for the variations mentioned above.

The following table lists the initial DCASE domain-specific terms and properties that were available to the developer for creating both the application specification and program piece specifications.

Table 4: DCASE domain terms and properties for the logistics domain

Domain Term	Description / Properties
logistics.trailer	The part of the truck that carries the goods. It includes properties like width, length, height, weight, type (e.g., refrigerator, tanker, flatbed, and box).
logistics.tactor	The driver compartment and engine of the truck. It has two or three axles. It includes properties like type (e.g., single, tandem), weight.
logistics.packing_list	List showing the **number** and **kinds** of items being shipped, and other information needed for transportation purposes.
logistics.item	Any unique manufactured or purchased part, material, intermediate, sub-assembly, or product. It includes properties like width, length, height, weight.
logistics.routing	Process of determining how shipment will move between origin and destination, including designation of carrier(s) involved, actual route of carrier, and estimated time enroute.
logistics.shipping_manifest	A document outlining the individual shipping orders included in a shipment. The manifest will show the reference number of each shipping order in the load, the weight and count of boxes or containers, and the destination.
logistics.order	A type of request for goods or services such as a purchase order, sales order, work order, etc. It includes properties such as item, quantity, and destination.

Based on the application requirements and the above domain terms and properties, the developer decomposed the application into three main components: assign items to trailers (knapsack packing problem), pair trailer to tractor for transport (assignment problem), and schedule delivery to destinations (vehicle routing problem). The application ISL specification, together with the specifications of these three components is shown in Figure 23.



```
piece: logistics_101
specification:
  requires:
    - logistics.order
    - logistics.tractor
    - logistics.trailer
  ensures:
    - logistics.routing
```

```
piece: Item2TrailerAssignment
specification:
  requires:
    - logistics.order
    - logistics.trailer
  ensures:
    - logistics.packing_list
implementations:
  python:
    code: file:first_fit.py
```

```
piece: Trailer2TractorAssignment
specification:
  requires:
    - logistics.packing_list
    - logistics.tractor
  ensures:
    - logistics.shipping_manifest
implementations:
  python:
    code: file:assignment.py
```

```
piece: VehicleRouting
specification:
  requires:
    - logistics.shipping_manifest
  ensures:
    - logistics.routing
implementations:
  python:
    code: file:routing.py
```

Figure 23: Order delivery ISL specification and relevant program piece specifications

During the development of the DCASE MVP, we devised initial approaches to addressing the following challenges:

- **Mapping application requirements to requires/ensures constraints**: For the two MVP applications, we defined domain-specific terms and properties that were used for expressing application requirements in terms of requires and ensures intent specifications. We anticipate that we will need to enhance our simple DSL used for requires and ensures constraints to allow the specification of more complex constrain expressions.
- **Expressing input/output requirements for applications and program pieces:** Developed applications and program pieces may require input/output to be formatted a certain way, e.g., delimited files. While it is relatively easy to capture high-level input/output formatting requirements using requires/ensures constraints (e.g., `logistics.input.type = "file" and logistics.input.format = "csv"`), being able to specify a specific schema for the file contents is more complicated, especially when no standardized schemas are available. We started developing simple utility program pieces that can transform one file format to a different format (e.g., JSON to csv). We plan to research whether required transformation could be synthesized automatically.
- **Reusable program pieces:** To avoid having developers implement "utility" program pieces repeatedly for different applications, we started researching techniques for supporting general-purpose and customizable program pieces. In particular, we started exploring design pattern and program pieces with holes. In our current implementation, program piece holes are represented using <%= … > notation, as shown in the `message_handler.yaml` example.
- **Start piece for Genetic Programming:** Traditional Genetic Programming techniques start from a randomly selected candidate pool and evolve it via mutations and crossover operations.

Approved For Public Release; Distribution Is Unlimited.

This approach may take a long time to find an acceptable solution, especially when the number of candidates is large. In DCASE, the candidate pool consists of program pieces, with each program piece having an intent specification associated with it in terms of the requirements it satisfies, expressed in requires/ensures constraints. Program piece specifications could be used for creating the initial candidate pool based on the application specification. For the DCASE MVP, we did that using a manual approach. We are currently investigating ways in which this can be automated.

- **Missing program pieces:** While we expect that a large percentage of application requirements will be satisfied by existing program pieces, Genetic Programming must handle cases where "missing" program pieces need to be introduced in order to satisfy all requirements. Developers would have to implement missing program pieces that satisfy specific *requires* and *ensures* constraints. The number of missing program pieces depends on both requires and ensures constraints in existing program pieces and the specific combination of these program pieces in the candidate solution pool. Selecting an "optimal" number of missing program pieces based on interactions between Genetic Programming and Symbolic Solving is an ongoing research challenge.
- **Better program synthesis:** In some domains, different algorithms may be available for achieving a specific goal. For example, there are several packing algorithms available (e.g., first fit, best fit, and next fit), with differences in their performance profiles. When multiple program pieces are available for solving the same problem (e.g., item packing), each implementing a different algorithm, selecting the "correct" program piece may be challenging when code quality or algorithm-specific properties are not included in the program piece specifications.
- **Program piece comparison:** In some domains, different algorithms may be available for achieving a specific goal. For example, there are several packing algorithms available (e.g., first fit, best fit, and next fit), with differences in their performance profiles. When multiple program pieces are available for solving the same problem (e.g., item packing), each implementing a different algorithm, selecting the "correct" program piece may be challenging when algorithm-specific properties are not included in the program piece specifications.

4.2 IDE Evolution

We went through two major iterations of IDE design. At first, we focused on text editor-based user interactions leveraging LSP and the DCASE Language Server. This approach became confusing to the user because we were overloading familiar IDE features with additional functionality that the user may not have expected. Furthermore, as IDE, Storage, and back-end component integration progressed, it became clear that overloading functions like "open file" and "save file" with essentially hidden features required explanation to the user. That defeated the purpose of an intuitive user interface.

We then pivoted to the notion of a DCASE toolbar we called "DCASE Features Window". This window was unobtrusive and pinned to the top of the user screen so it was both out of the way and easily accessible. Another advantage to this pivot was that DCASE features became more definite and clear-cut to the user instead of opaque functions hiding behind the IDE text editor. We still used LSP to provide functionalities like auto-complete, capturing text changes, and highlighting text. We also used the DCASE Language Server to bring up and tear down the DCASE Features Window as well as process toolbar interactions. For cross-platform portability and easy integration with LSP, we chose to implement the Features Window in Java Swing and FX.

4.2.1 First Iteration: Text Editor Features

For the first iteration, we put all the DCASE features into the ISL file. Whenever the user opened a new or existing file with the '.isl' extension, the LSP tied file operations to the DCASE Language Server. We implemented the following features:

- On opening an '.isl' file, a welcome message that directed the user to DCASE references (which we intended to turn into a tutorial later in the program).
- Auto-complete for existing domain terms and properties, with documentation, shown in Figure 24.

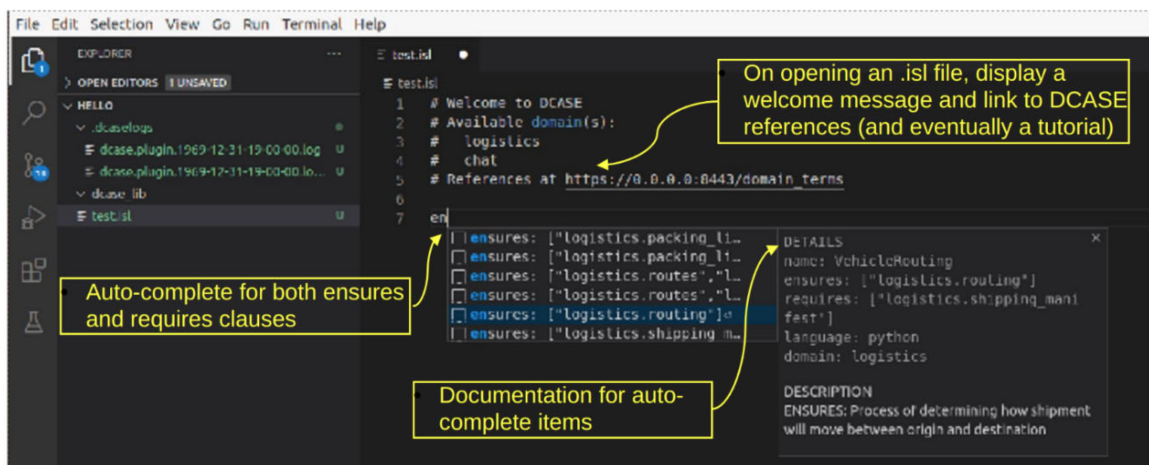
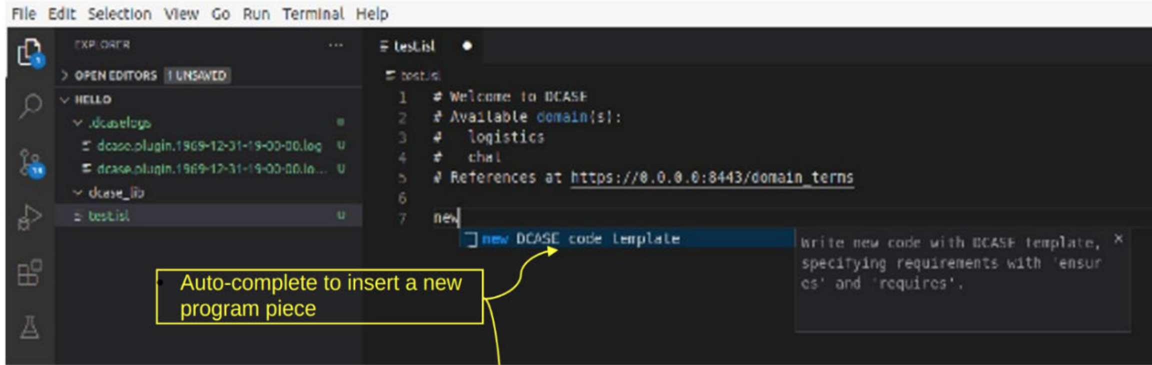


Figure 24: DCASE ISL file example

- Auto-complete could also insert a new program piece template for the user to fill out when she wished to create a new program piece (see Figure 25). Of course, the user can put in as many requires and ensures clauses and program pieces in the ISL file. In this approach, everything with which the user interacts - requires and ensures clauses as well as program pieces - exist in the application ISL file (see Figure 26).



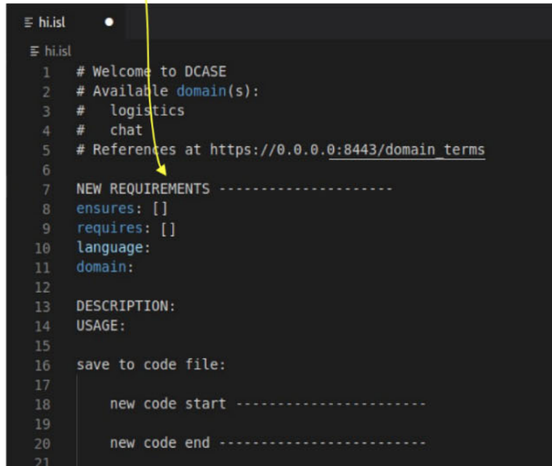


Figure 25: Auto-complete for new program piece template

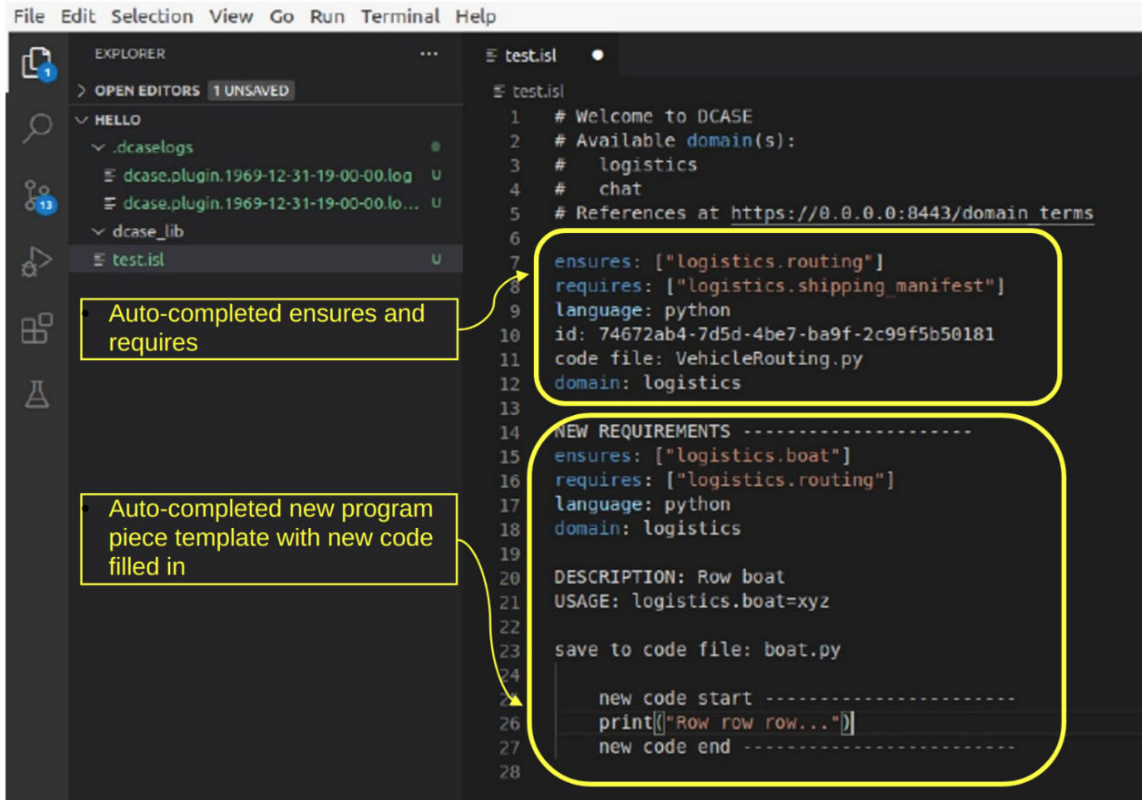


Figure 26: Auto-complete with existing and template program pieces

- Saving the file, whether by IDE 'Save File' menu option or 'ctrl-s' keyboard shortcut, pushed new program piece(s) to DCASE Storage and triggered the Genetic Programming to recommend existing program pieces available via Storage. All relevant program pieces were eventually deposited into the DCASE library directory in the IDE. From here, the user could proceed to run and debug the code in the DCASE library directory using the IDE's native 'Run' and 'Debug' mechanisms.

While we were able to leverage LSP and TextMate to provide DCASE functionality via familiar features such as open file, auto-complete, save file, and text coloring, the user interactions with the ISL file turned out to be complicated. Here were the challenges the user and we faced:

- Putting application requires and ensures plus program piece code in the same ISL file made the file difficult to read and manipulate.
- If the DCASE Language Server removed the program pieces upon the user saving an ISL file to their own program files with appropriate extensions so the IDE to run them later (e.g. '.py', '.java', or '.ts' for Python, Java, or Typescript, respectively), the program pieces "magically disappear" from the user's point of view. This created quite a bit of confusion and required more extensive explanation than we liked.
- If the user broke up her application into multiple '.isl' files for readability, then the language server would have to track multiple application ISL files, and this put the onus on the user to monitor all the relevant ISL files along myriad of program piece files deposited in the library directory.

- ‘Save File’ function was too overloaded. From the user’s perspective, not only did it save what she typed into a local file system, it also
    1. Pushed the file to DCASE Storage (done invisibly)
    2. Deposited program pieces into the library directory (visible only if the user is paying attention)
    3. Program pieces “disappear” from the ISL file (visually jarring and confusing)
    4. The user may not have meant to save the file, and so potentially incomplete or erroneous code has been pushed to DCASE Storage (and it disappeared from the ISL file)
    5. It was not clear to the user she could edit the corresponding code file in the library directory
    6. When all has been written, it was not clear to the user what to do next

4.2.2 Second Iteration: DCASE Features Window

To address these challenges, we created a DCASE toolbar (DCASE Features Window, Figure 27) that could guide the user through writing, building, and running an ISL application. This Features Window was compact and easily tucked away to the top of the screen while remaining accessible to the user whenever she needed to use it.

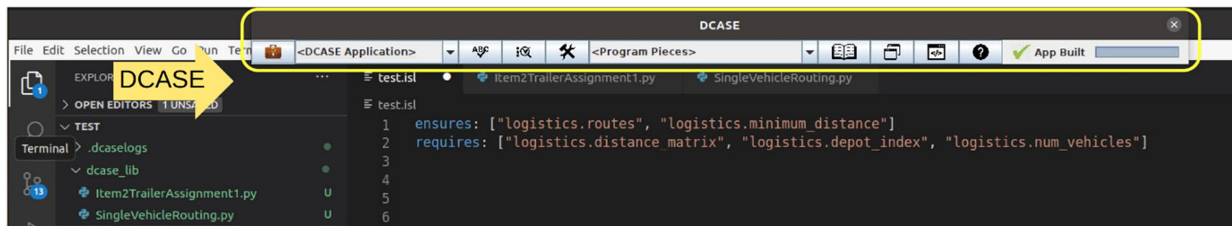


Figure 27: DCASE Features Window pinned to the top of the user screen

The toolbar offered three types of intuitive graphical elements:

- Icon buttons with tooltips, which explain what the button did.
- Clearly labeled drop down menus specifically designed for the user to work with application ISL file versus program pieces.
- Status bar that informed the user what DCASE was doing.

The Features Window has been divided into four areas (see Figure 28):

1. Application ISL functions for the user performing the role of a Subject Matter Expert
2. Program piece development area for the Subject Matter Expert with software development experience
3. Browsing references
4. DCASE status display

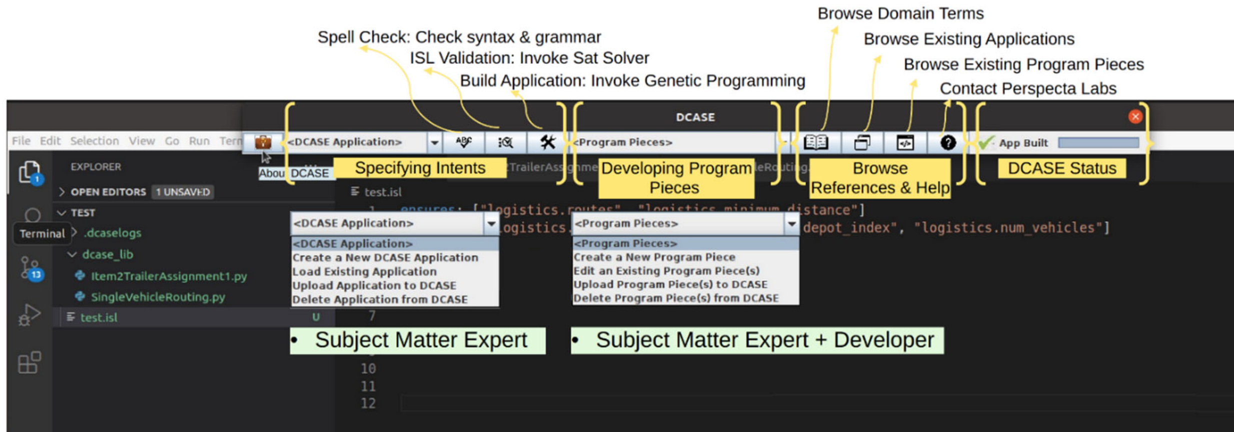


Figure 28: DCASE Features Window

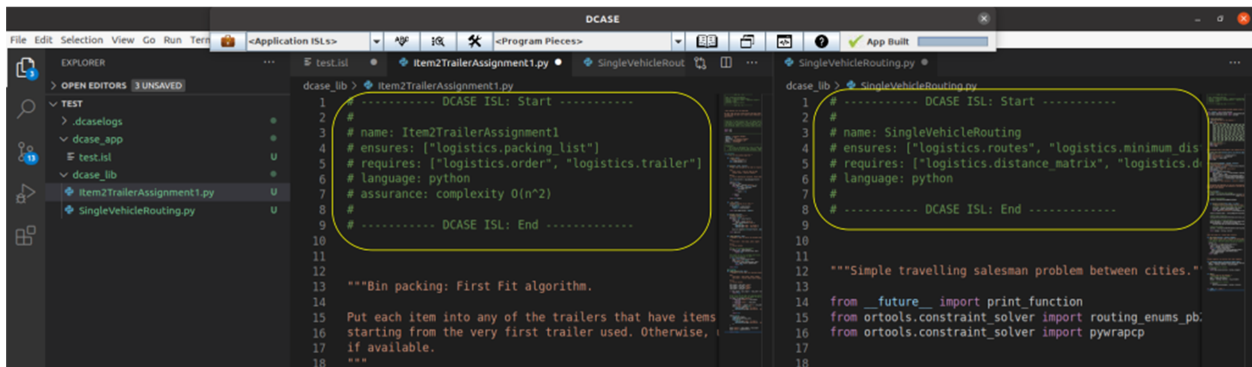


Figure 29: DCASE program pieces have embedded meta-data for tracking provenance

When DCASE generated a solution based on application requirements and existing program pieces, the resulting source code could be viewed and edited, if required, using the IDE. Program piece specifications and the application specification were still incorporated into the generated code using programming language specific comment segments (Figure 29). In this way, DCASE tracked provenance information about intent specifications and was able to identify the impact of requirements changes in the generated application.

We also leveraged the default browser to help the user examine DCASE references or contact Perspecta Labs for assistance (Figure 30).

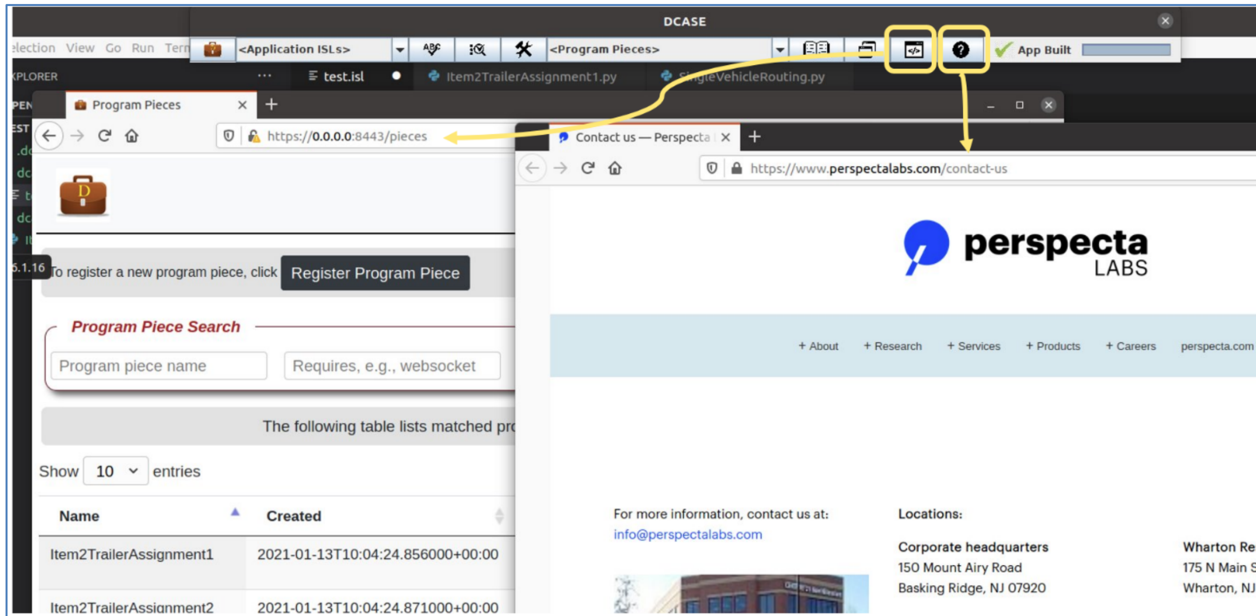


Figure 30: DCASE Features Window can invoke the default browser

This DCASE toolbar addressed challenges from our initial text editor-based iteration as follows:

- Put all requires and ensures clauses into single application ISL, and the ISL file could only contain requires and ensures clauses plus comments. It no longer housed any program piece implementation.
- Program piece implementations, generated or new, were filed under the DCASE library directory. The user worked with program piece files directly from this library.
- The toolbar implicitly guided the user on how to navigate DCASE - roughly from left to right.
- 'Save File' was no longer overloaded. All the additional DCASE functionalities have been moved to the Features Window. Likewise, for 'Open File'.

While we redesigned a majority of how DCASE functionalities interfaced with the user, we kept two major features from our previous iteration: Auto-complete with documentation and depositing program pieces into DCASE library directory after synchronizing with DCASE Storage and back-end components (Figure 31).

Underneath the hood, our LSP based DCASE Language Server was still providing services like auto-complete along with creating/tearing down the DCASE Features Window and handling user interface requests when she interacts with the toolbar.

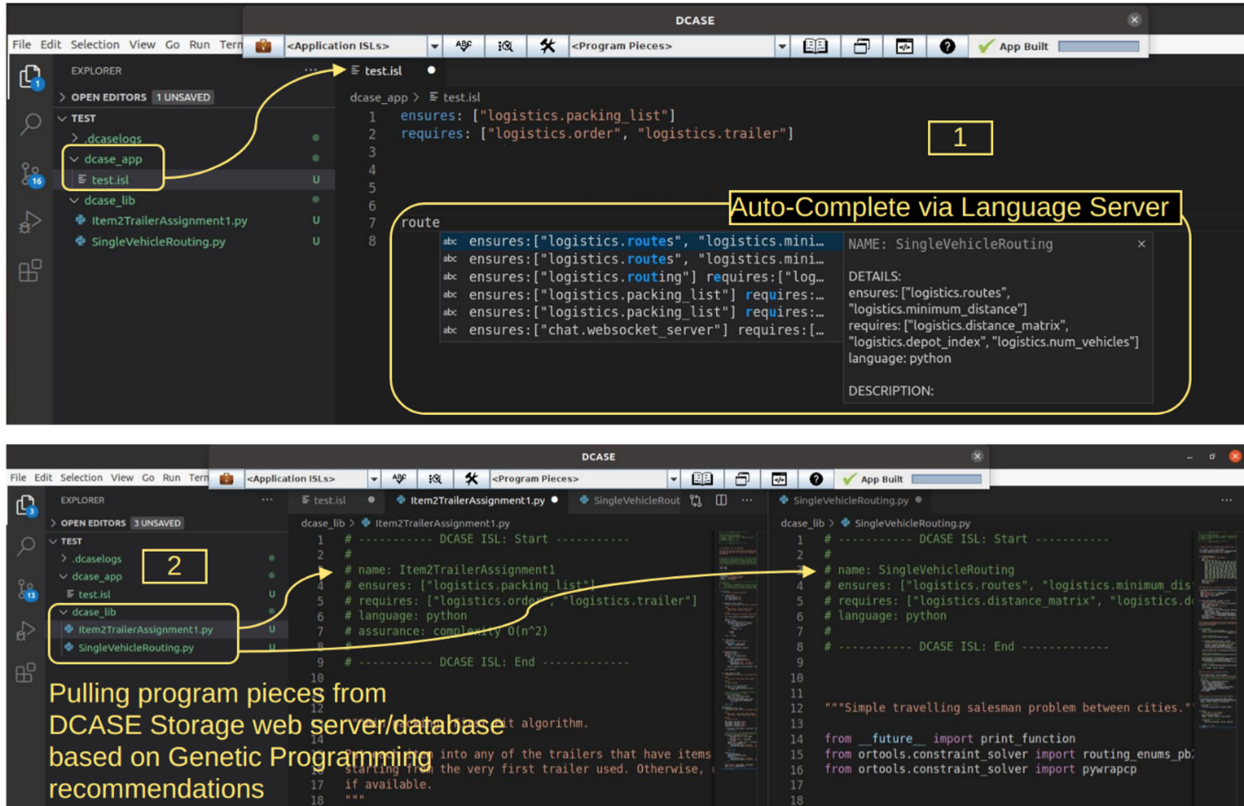


Figure 31: Features kept from the initial iteration

4.2.3 Running DCASE Generated Code

As much as possible, we wanted our DCASE user to use IDE functions that were already available in the IDE. We envisioned using the Language Server's "file saved" response to trigger Genetic Programming, pull the available and appropriate program pieces from the DCASE database, and deposit them into local DCASE library directory.

For example, the user can take the following steps to have DCASE generate code (illustrated in Figure 32):

1. Write/update an application ISL file
2. Save ISL file and thereby
    a. Triggers syntax and grammar check
    b. Which then triggers DSL validation
    c. Which then runs Genetic Programming
    d. Which finally deposits recommended program pieces into a DCASE library directory
3. Write new program pieces, if applicable, and save the program pieces files – this pushes the new program pieces to both DCASE Storage and DCASE library directory
4. Use the IDE's "Run" menu to configure and run code in the DCASE library directory

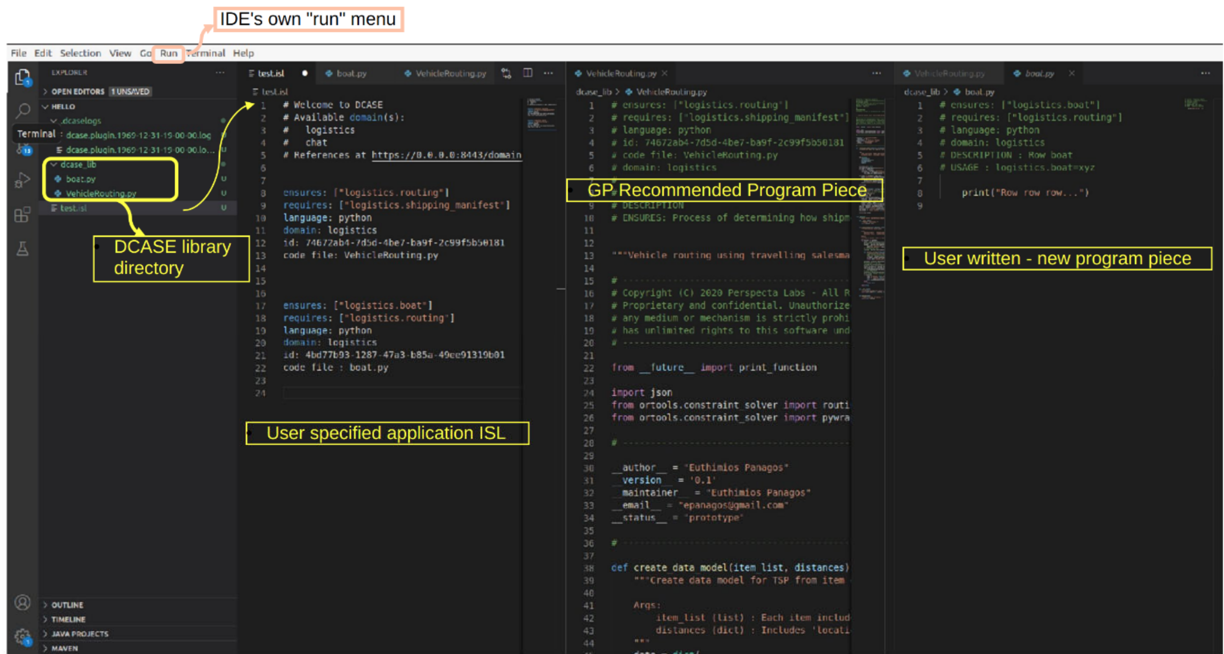


Figure 32: Generated application example

4.3 Performant Embedded Systems

During the last few months of the IDAS program, we shifted our attention from the logistics domain to that of embedded systems and system of systems. As part of this shift, we focused on applying DCASE to the following two use cases: an elevator controller and autonomous systems. For the former, we used a notional multi-floor elevator installation. For the latter, we used the OpenUxAS open-source platform (https://github.com/afrl-rq/OpenUxAS) as the target platform for enhancing existing services and developing new services using DCASE.

4.3.1 Elevator System

Our goal in applying DCASE to an elevator system was to assess whether the DCASE **program piece** concept could be applied to a combination of software and hardware components with both functional and non-functional constraints. As part of this effort, we created program piece specifications (no implementation code) to represent the following software and hardware components of a typical elevator system:

- Central *Controller* unit;
- Different types of elevator cars and doors (*LowLatencyCar*, *HighSpeedCar*, *LowLatencyDoor*, *LargeWidthDoor*);
- Different types used by the mechanic parts of the elevator system (*RegularChip*, *LowLatencyChip*).

Using the above program pieces, we created application specifications with different *requires* and *ensures* constraints associated with car door sizes and speeds. We then applied our Generic Programming and Symbolic Solving to these specifications and were able to find candidate

solutions, confirming that the DCASE approach can be easily applied to different domains. One of these solutions is shown in Figure 33.

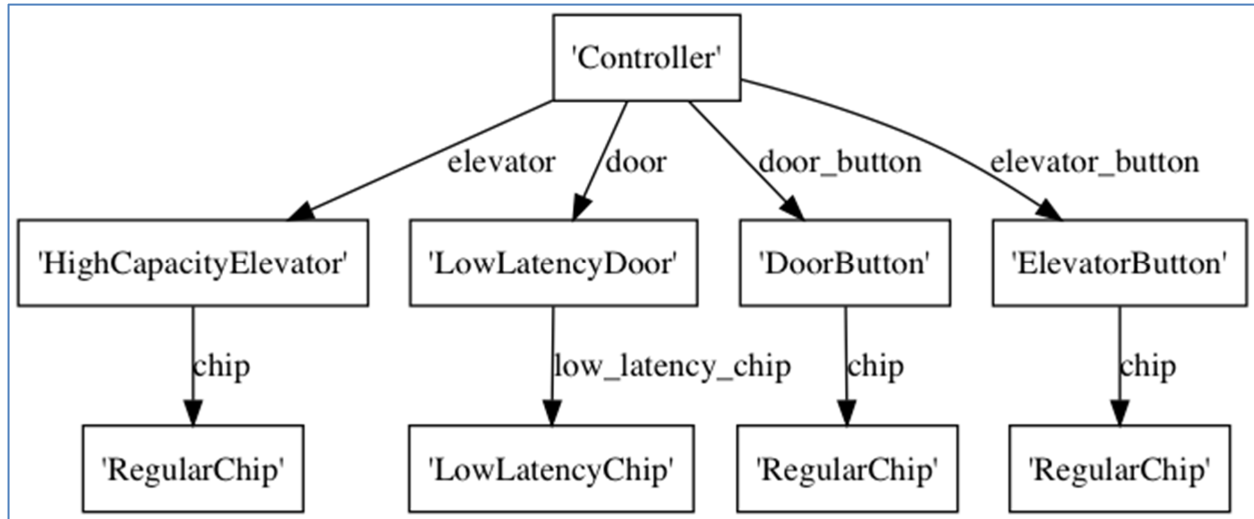


Figure 33: DCASE solution to an elevator system application

4.3.2 OpenUxAS

We started by reading the Unmanned Systems Autonomy Services (UxAS) documentation in detail, and then picking one example mission, *WaterwaySearch*, and delving deeply into its organization. In particular, we looked at the initial configuration settings for WaterwaySearch and the orchestration of messages sent between UxAS components to carry out WaterwaySearch. Next, we looked at another example mission, *DistributedCooperation*, to understand the similarities and differences between it and WaterwaySearch. Our goal was to find a set of changes that would adapt the WaterwaySearch mission setup into the DistributedCooperation mission setup. The idea was that then we would develop a set of program pieces that could express those differences, so that we could get the general setup for one mission or the other by changing the `requires` clause of a root program piece that depended on other pieces in that set.

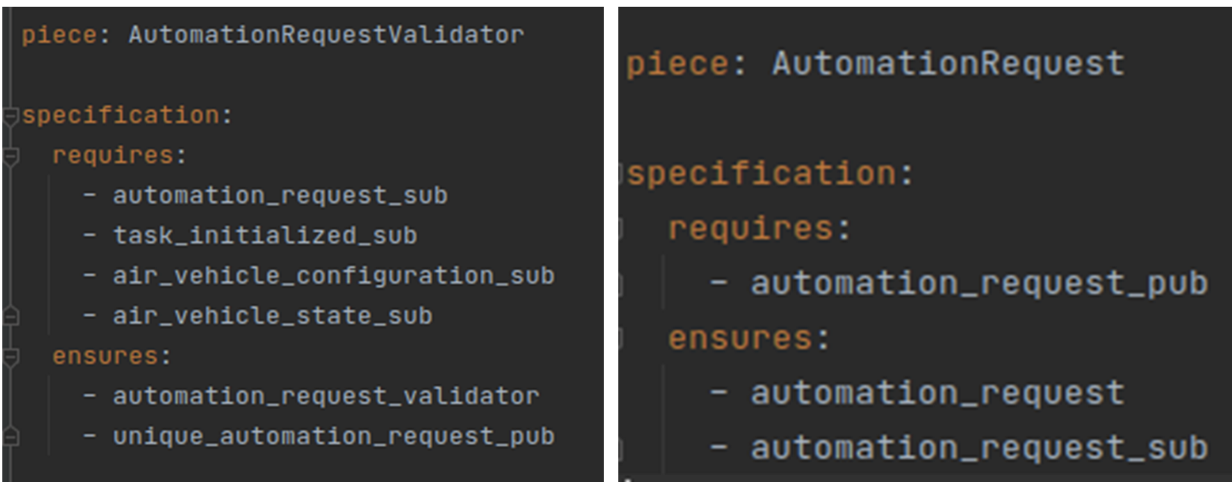
We also explored applying the DCASE system to upgrading components of the software. These components included the ZMQ message bus, the Lightweight Message Construction Protocol (LMCP) message format, and the Zyre distributed broker. We evaluated the software engineering effort it would take to replace ZMQ with RabbitMQ. One of the difficulties we ran into was that the Zyre distributed broker is tied to ZMQ and RabbitMQ did not have a clear associated distributed broker. We did consider federation, shovels, and clustering but no solution seemed to work as well as Zyre in a peer-to-peer environment where peers regularly come and go. For LMCP, we considered a simpler JSON format and XML Unfortunately, none of the upgrade solutions benefited by our Genetic Programming / Symbolic Solving solution, which excelled with many options. For us, our message queue upgrade and messaging format upgrades only had a few options.

Finally, we focused our efforts on developing a new service that required the creation of a new message type. The following section provides details about this effort.

4.3.2.1 *OpenUxAS Waterway Search*

We developed program pieces that represent both the services and messages as program pieces for the Waterway Search task (pre-defined in their repo). The system includes a *Root* piece that requires all of the services and messages, and therefore acts as a specification for the entire system. Each service piece requires the messages that it is subscribed to and ensures the service itself as well as any messages that it publishes. For example, the *AutomationRequestValidator* piece requires `automation_request_sub` (it is subscribed to the *AutomationRequest* message), and it ensures `automation_request_validator` (connection to *Root* piece) and `unique_automation_request_pub` (it publishes the `UniqueAutomationRequest` message).

Table 5: Sample OpenUxAS program piece specifications



```
piece: AutomationRequestValidator

specification:
  requires:
    - automation_request_sub
    - task_initialized_sub
    - air_vehicle_configuration_sub
    - air_vehicle_state_sub
  ensures:
    - automation_request_validator
    - unique_automation_request_pub
```

```
piece: AutomationRequest

specification:
  requires:
    - automation_request_pub
  ensures:
    - automation_request
    - automation_request_sub
```

Each message piece requires its own publication and ensures the message itself as well as the subscription to that message. For example, the *AutomationRequest* message piece requires `automation_request_pub`, and ensures `automation_request` and `automation_request_sub`. This design allows the program pieces to reflect the pub-sub architecture of OpenUxAS. A service piece is not satisfied unless its subscribed messages have been published, and once it is satisfied, it ensures the messages that the service publishes. Likewise, a message piece is not satisfied unless a service published that message, and once the message is published, it ensures the services that subscribe to it. In addition, since both the service and message pieces are connected to the *Root* piece through a constraint, our GP algorithm can discover all these pieces by starting at the *Root* piece in its graph-based search, shown in Figure 34.

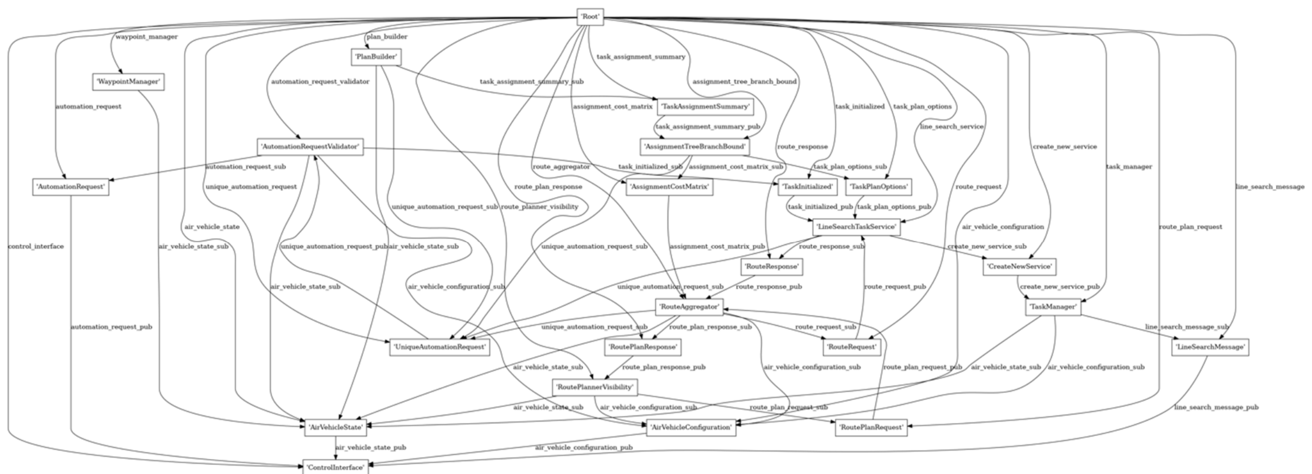


Figure 34: GP-based discovery of program piece connections

The next step was to develop program pieces that not only consisted of XML implementation, but also source code. We started with an example of how to add a new field in a message, and then for the subscriber to print that it read the field. This required extending the program pieces to implement both XML and C code that was emitted when DCAS-GP found a solution.

4.3.2.2 ***Embellishing the UxAS*** *HelloWorld* ***Example***

In this section, we describe the modification of an existing service, the creation of a new message type, the creation of a new service that works with the existing service, and the creation of a new service orchestration that integrates the two service types. While OpenUxAS deals with autonomous vehicles and services appropriate to that space, to explore the mechanics of the activities just listed, it suffices to take as a base the generic *HelloWorld* example available with the source distribution of OpenUxAS.

4.3.2.2.1 Initial Scenario

The `01_HelloWorldExample` runs two copies of the same service, `HelloWorld`. Each service instance is configured with a single message to send and the period with which it sends the message. These parameters are usable by the service only because it has been coded to understand them. For instance, the `HelloWorld` service contains this code:

```
#define STRING_XML_STRING_TO_SEND "StringToSend"

bool HelloWorld::configure(const pugi::xml_node& ndComponent) {
    // process options from the XML configuration node:
    if (!ndComponent.attribute(STRING_XML_STRING_TO_SEND).empty() {
        m_stringToSend = ndComponent.attribute(STRING_XML_STRING_TO_SEND).value();
    }
    …
}
```

The provided example contains two services that operate independently, in the sense that they do not respond to each other's messages. In their `initialize()` method, they each set a timer that will fire at the configured frequency. The timer is provided with a callback to the

OnSendMessage(), which is responsible for sending the configured StringToSend. While the StringToSend denotes the message contents, the actual message that is sent via the pub-sub infrastructure is established within OnSendMessage():

```
auto keyValuePairOut = std::make_shared<afrl::cmasi::KeyValuePair>():
keyValuePairOut->setKey(std::to_string(m_serviceId));
keyValuePairOut->setValue(m_stringToSend);
sendSharedLmcpObjectBroadcastMessage(keyValuePairOut);
```

The CMASI.xml file defines the message type, KeyValuePair, in the namespace afrl.cmasi. This message contains two strings, a key and a value. OnSendMessage() uses the sender's service_id as the key, with the configured StringToSend as the value. The message is broadcast to all services that subscribe to that message type. The HelloWorld service establishes a subscription to listen for afrl::cmasi::KeyValuePair messages in its configure() method, as so:

```
addSubscriptionAddress(afrl::cmasi::KeyValuePair::Subscription);
```

A service can subscribe to receive messages of multiple types. In its processReceivedLmcpMessage() method, it receives a generic LMCP message and then first determines the specific type of the message before proceeding to process it. The code then casts the generic message to the specific type and accesses the fields of the message through getters.

```
bool HelloWorld::processReceivedLmcpMessage
(std::unique_ptr<uxas::communications::data::LmcpMessage>  receivedLmcpMessage) {
    if (afrl::cmasi::isKeyValuePair(receivedLmcpMessage->m_object)) {
                auto keyValuePairIn =
std::static_pointer_cast<afrl::cmasi::KeyValuePair> (receivedLmcpMessage-
>m_object);
              std::cout << "*** RECEIVED:: Received Id[" << m_serviceId << "] Sent
Id[" << keyValuePairIn->getKey() << "] Message[" << keyValuePairIn->getValue() <<
"] *** " << std::endl;
    }
    return false;
}
```

4.3.2.2.2 Modifications

This basic example was modified in various ways:

1. The services were configured so that they would each send their greeting in a different language. This involved simply changing the configuration file to have one service send the greeting in English and one in German.
2. The services no longer used a timer to dictate when they sent messages. Instead, they each sent one message at startup. After that point, they would reply whenever they received a message. The timer setup was removed from the initialize() method, and the start() method was changed to send an initial message. Furthermore, the

processReceivedLmcpMessage() method was set up to invoke OnSendMessage whenever a service subsequently received a message.

3. To keep the services from getting into a loop of constant acknowledgments, it was decided to mark the messages with a parameter to denote whether the message was an acknowledgment of a previous message. Services would then be programmed to take no action in response to a message that was an acknowledgment. In addition, with an eye toward introducing a translation service, the messages would also carry details about the country of origin of the sender, as well as the language of the message contents.

   A new message type was introduced in a new XML file in the mdms folder, which contains the schemas for the messages. This file was called TEXTCOMM.xml:

```
<MDM>
    <SeriesName>TEXTCOMM</SeriesName>
    <Namespace>uxas/messages/textcomm</Namespace>
    <Version>1</Version>
        <Comment>Simple Text Messages </Comment>
        <EnumList>
            <Enum Name="Country">
                <Entry Name="US" />
                <Entry Name="DE" />
            </Enum>
        </EnumList>
        <StructList>
            <Struct Name="TextMessage">
                <Field Name="SenderID" Type="int32" Default="0"/>
                <Field Name="SenderCountry" Type="Country" />
                <Field Name="Contents" Type="string"/>
                <Field Name="ContentsCountry" Type="Country" />
                 <Field Name="AckStatus" Type="bool" Default="false" />
                  </Struct>
        </StructList>
</MDM>
```

   Additionally, a new parameter was added to the configuration xml file, to allow the specification of the service's country, so that it could be put into the message contents. (Also, a user-specified service id was also added to the configuration, to help keep the services straight during a run.) The new configuration looked like:

```
<UxAS EntityID="100" FormatVersion="1.0" EntityType="None"
      RunDuration_s="100.0">
    <Service Type="HelloWorld" ID="1" Country="DE"
             StringToSend="Guten Tag"/>
    <Service Type="HelloWorld" ID="2" Country="US"
             StringToSend="Hello"/>
</UxAS>
```

4. The HelloWorld configure() needed to be modified to handle the current set of configuration parameters.
5. To enable HelloWorld to use the new TextMessage definition, CPP classes associated with TEXTCOMM.xml needed to be added to the LMCP library that is linked with the services

when they are built. The utility OpenUxAS/resources/RunLmcpGen.py was invoked to create the library from all the XML files, including TEXTCOMM.xml, in the mdms folder. The generated library classes allow for the construction and parsing of messages and their serialization/deserialization to/from the message platform.

6. HelloWorld then needed to be coded to use the new message type instead of the one it had been using. First, it needed to subscribe to the new message in its configure() method:

```
addSubscriptionAddress(uxas::messages::textcomm::TextMessage::Subscription);
```

The `OnSendMessage()` method needed to be changed to create a message of this type. It also needed to be changed to fill in the fields of this message. This code snippet shows the creation of a TextMessage and the setting of the sender's country from the internal string variable, m_country:

```
void HelloWorld::OnSendMessage(bool initial) {
    auto tm = std::make_shared<uxas::messages::textcomm::TextMessage>();
    tm->setSenderCountry(uxas::messages::textcomm::Country::get_Country
(m_country));
```

Likewise, the `processReceivedLmcpMessage()` method needed to be altered to look for a message of type TextMessage and parse its contents. This code shows that the method looks for TextMessage, and when it finds it, it looks at fields of the message to determine if it should call `OnSendMessage()` to send a message.

```
bool HelloWorld::processReceivedLmcpMessage(
std::unique_ptr<uxas::communications::data::LmcpMessage>
receivedLmcpMessage) {
if (uxas::messages::textcomm::isTextMessage(receivedLmcpMessage->m_object)){
       auto tm =
std::static_pointer_cast<uxas::messages::textcomm::TextMessage>
(receivedLmcpMessage->m_object);
             if (!tm->getAckStatus() && (tm->getSenderID() != m_id)  &&
       (tm->getContentsCountry() ==
uxas::messages::textcomm::Country::get_Country(m_country))){
                    std::cout << m_id<<" *** RECEIVED A MESSAGE " <<tm-
>getContents()<<" from "<<tm->getSenderID()<<" with ack status = "<<tm-
>getAckStatus()<<" so sending an ack"<< std::endl;
                    this->OnSendMessage(false);
             }
    }
    return false;
}
```

7. The preceding code shows that a HelloWorld service is now coded to respond to a message it receives if these conditions hold:
   a. The received message is not an acknowledgment of a previous message.
   b. The received message is not one that it originated.
   c. The nationality (language) of the message contents are its own language.

The last restriction means that the two services in the current scenario will not acknowledge their peer's greeting, because it is in the wrong language.

8. To get past this point, a new service, `HelloTranslator`, was created. The service would also establish a subscription for messages of type `TextMessage`. It would be coded to respond to every `TextMessage` it received. The message it sent in response, though, would simply replace the contents and the contents country with those of the other service. So, upon receiving the following message:

```
<TextMessage>
	<SenderID>1</SenderID>
	<SenderCountry>US</SenderCountry>
	<SenderContents>Hello</SenderContents>
	<ContentsCountry>US</ContentsCountry>
	<AckStatus>false</AckStatus>
</TextMessage>
```

It would send this message in response:

```
<TextMessage>
	<SenderID>1</SenderID>
	<SenderCountry>US</SenderCountry>
	<SenderContents>Guten Tag</SenderContents>
	<ContentsCountry>DE</ContentsCountry>
	<AckStatus>false</AckStatus>
</TextMessage>
```

The German HelloWorld service, which does not process a message that comes to it directly from the US Hello service, does process this translated message and send a response. The US HelloWorld service, for its part, will not process the response coming directly from the German HelloWorld Service, but it will process the message that the `HelloTranslator` service puts on the wire after it translates the German service's acknowledgment message.

It should be noted that creating a new service in OpenUxAS is quite easy. A script called `OpenUxAS/src/cpp/Services/00_NewService.py`, when given the name of the new service, generates a skeleton cpp and header file for the service, and includes a reference to this code in the build script. The skeleton code contains the methods already listed in this document, methods like `configure()`, `initialize()`, `start()`, and `processReceivedLmcpMessage()`. They simply need to be fleshed out with code similar to the code found in the HelloWorld service.

9. The last thing that remained was to create a new service orchestration scenario that includes the new service. It is identical to the previous file (item 3 above), with the addition of a single line:

```
<UxAS EntityID="100" FormatVersion="1.0" EntityType="None"
      RunDuration_s="100.0">
    <Service Type="HelloWorld" ID="1" Country="DE"
             StringToSend="Guten Tag"/>
    <Service Type="HelloWorld" ID="2" Country="US"
             StringToSend="Hello"/>
    <Service Type="HelloTranslator" />
</UxAS>
```

Both of these scenarios can still be run. When the one that omits the `HelloTranslator` is run, each `HelloWorld` service sends its initial greeting in its own language, but there are no acknowledgement messages sent. On the other hand, when the scenario employing this latest configuration file is used, the `HelloTranslator` is inserted into the scenario, and acknowledgement messages are generated in response to the message translations.

4.3.2.3 OpenUxAS Observations

Due to the cancellation of the program, we stopped pursuing this effort before finishing it. However, we did learn several important things about OpenUxAS. First, as a framework, UxAS makes heavy use of software engineering patterns and idioms to yield a flexible, adaptable, component-based framework. Second, OpenUxAS pays little or no attention to resource usage; essentially all of the system components are available at all times, and there are layers and layers of abstractions that add a lot of overhead in the name of flexibility. Finally, OpenUxAS example missions themselves do not really come with code. Rather, they are XML configuration files that specify the OpenUxAS components they need; the initial mission configuration; and the mission goals.

Based on these observations, in retrospect, it seems like OpenUxAS it not a good example of a resource-constrained embedded system that is "exquisitely engineered." Rather, the OpenUxAS developers have carefully considered in what dimensions UxAS missions need flexibility and changeability, and they have engineered OpenUxAS to support those changes by creating many, appropriate abstraction barriers. To be a good target for DCASE, we would need to find design choices inside of OpenUxAS for which the developers had *not* anticipated change, and which could not support the added overhead and complexity of introducing additional abstractions.

4.4 End-to-end integration

The IDAS team did work towards DCASE end-to-end integration of all developed components. The DCASE integrated architecture, shown in Figure 35, illustrates the workflow from the IDE through the Controller / Storage component, then analyzed by the GP / Symbolic Solving component after which results are returned to the IDE through the Controller / Storage component.

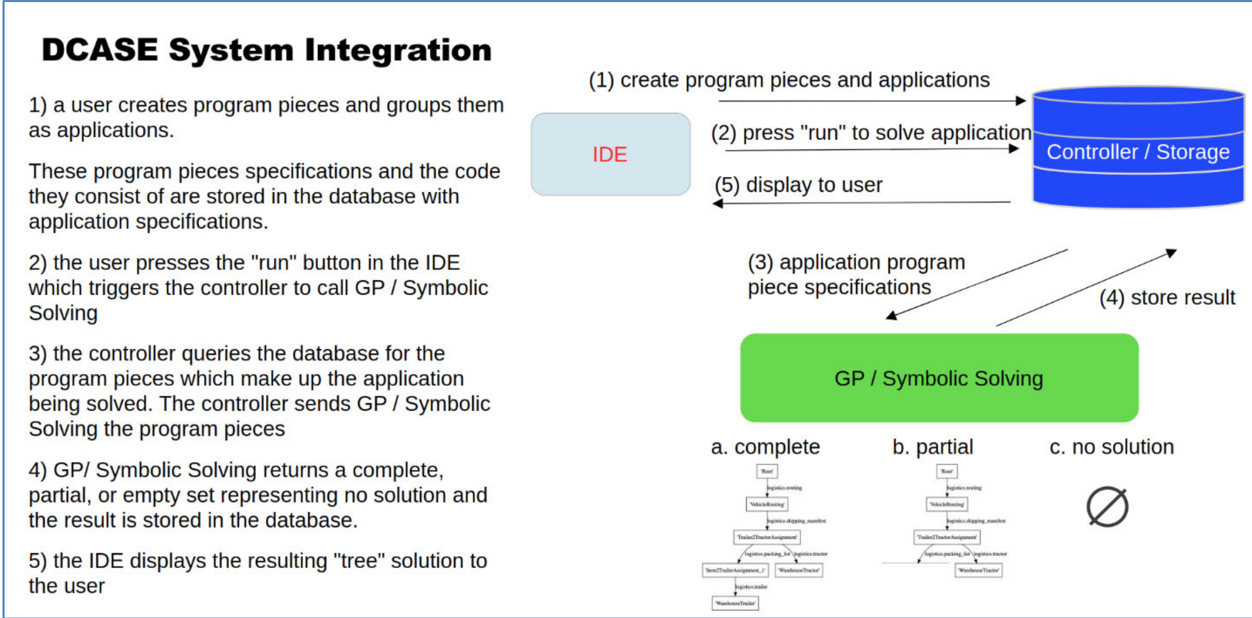


Figure 35: DCASE end-to-end component integration

The integration work that occurred before program end was between the GP / Symbolic Solving component and the Controller / Storage component. The Controller / Storage component contains an admin console where application YAML specification files can be uploaded. Program piece specification can also be uploaded via the IDE or the admin console. A "Run GP" button in the application YAML section of the admin console, when pressed, triggers the Controller / Storage component to send program pieces to the GP / Symbolic Solving component. In the current design, the GP / Symbolic Solving component exists on the same machine as the Controller / Storage component but this does not have to be the case. When the GP / Symbolic Solving component receives program pieces, it finds a solution with those pieces. It may include hypothetical or partial program pieces and returns a link to the folder containing the PNG and JSON solutions. In the current integration, a gallery viewer can view the PNG images from GP / Symbolic Solving.

5.0 CONCLUSIONS

DCASE (Deferred Concretization Adaptive Software Environment) automates the software sustainment loop using a novel combination of techniques with proven scalability and program-generation speed, while embracing software engineering concepts that ensure ease of use by traditional developers.

The Intent Specification Language (ISL), a key element of our solution, enables a developer to express the problem to be solved. The programmer uses the ISL to create an intent specification that describes the program's structure in terms of reusable, domain-specific abstractions, and the program's semantics in terms of (1) relationships among abstraction elements, (2) constraints on computational resources, execution environment, performance, and security, and (3) (optionally) symbolic test cases that express the program's expected behavior at an abstract level. Developers use the ISL inside an Integrated Development Environment (IDE) that embodies the ISL and a catalog of **program pieces**, reusable module that include:

- An abstract specification describing (in ISL) the structure and semantics of the piece;
- A concrete part with implementation(s) in one or more target languages (e.g., Java or Python); and
- Assurance evidence for each implementation (e.g., validated via testing or with a formal proof).

DCASE automates program generation (the how) based on the intent specification (the what) by synergistically combining two program synthesis techniques: **Genetic Programming** and **Symbolic Solving**. Genetic Programming explores the problem search space identified by the intent specification and combines existing program pieces to generate candidate solutions (i.e., abstract, ISL programs) that are structurally correct, but which may contain pieces with unbound parameters and may not satisfy all intent constraints. Symbolic Solving assesses each candidate's functional (i.e., semantic) fitness by constructing and solving the conjunction of (1) constraints from the intent specification and (2) constraints implied by the candidate's constituent pieces.

If there is no assignment to parameters that can satisfy all the intent constraints, Symbolic Solving provides to Genetic Programming a measure of the candidate's functional fitness, thus helping to bias the evolution of future generations of candidates toward variations that are more likely to satisfy the intent specification. When Symbolic Solving finds a solution, Code Generation uses the candidate's constituent program pieces to emit code in a target programming language. In addition, Evidence Generation produces an assurance argument comprised of evidence for the solution's constituent program pieces and their correct composition (i.e., via pre-/post conditions), the selected parameters, and a representation of the intent constraint satisfied by the solution.

We completed a steel-thread prototype to demonstrate the DCASE concepts. We used the prototype to build a chat application server and then apply four subsequent adaptations to the application. This exercise demonstrated the feasibility of the DCASE approach and highlighted areas for further research. With further investment, we believe the DCASE tool can become a practical tool that will greatly speed up and reduce costs in maintaining and adapting existing or future DoD software systems.

6.0 REFERENCES

[BO16] Burns, B., and Oppenheimer, D. Design patterns for container-based distributed systems. In Proceedings of the eighth USENIX Workshop on Hot Topics in Cloud Computing (Hot-Cloud 16), 2016.

[CR06] Lori A. Clarke and David S. Rosenblum, *A historical perspective on runtime assertion checking in software development* in: ACM SIGSOFT Software Engineering Notes 31(3):25-37, 2006.

[GJ+07] Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., & Willcock, J. (2007). An extended comparative study of language support for generic programming. *Journal of Functional Programming, 17*(2), 145-205.

[GV+94] Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Massachusetts, 1994.

[HC01] George T. Heineman, William T. Councill (2001). Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Professional, Reading 2001 ISBN 0-201-70485-4.

[HW+04] Hohpe, G., Woolf, B., Brown, K., and Fowler, M. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2004.

[S08] Armando Solar-Lezama. Program Synthesis by Sketching. PhD thesis, EECS Dept., UC Berkeley, 2008.

[SR+05] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, Kemal Ebcioglu. Programming by sketching for bit-streaming programs. PLDI '05 Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Chicago IL, June 2005.

7.0 LIST OF ACRONYMS

AFRL	Air Force Research Laboratory
AML	Algebraic Modeling Languages
API	Application Programming Interface
MQ	Asynchronous Messaging Library
CPP	C Plus Plus; C++
CSS	Cascading Style Sheet
CSV	Comma Separated Values
CEGIS	counter-example-guided inductive synthesis
DB	Database
DCASE	Deferred Concretization Adaptive Software Environment)
DOD	Department of Defense
DAG	Directed Acyclic Graph
EBNF	Extended Backus–Naur Form
XML	Extensible Markup Language
GECCO	Genetic and Evolutionary Computation Conference
GP	Genetic Programming
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IRC	Internet Relay Chat
ISL	Intent Specification Language
JSON	JavaScript Object Notation
LSP	Language Server Protocol
LMCP	Lightweight Message Construction Protocol
MVP	minimum viable product
OS	Operating System
PEGs	Parsing Expression Grammars
PNG	Portable Network Graphics
RPC	Remote Procedure Call
REST	Representational State Transfer
SMT	Satisfiability Modulo Theory
SV	Solution Viewer
UxAS	Unmanned Systems Autonomy Services
VRP	Vehicle Routing Problem
VS	Visual Studio
YAML	YAML Ain't Markup Language
XML	Extensible Markup Language
ZMQ	Zero MQ