



AFRL-RI-RS-TR-2021-164

INTERFACES, MODELS, AND MONITORING FOR RESOURCE-AWARE TRANSFORMATIONS THAT AUGMENT THE LIFECYCLE OF SYSTEMS (IMMORTALS)

RAYTHEON BBN TECHNOLOGIES CORP.

SEPTEMBER 2021

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

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 the 88th ABW, Wright-Patterson AFB 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-2021-164 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

STEVEN L. DRAGER
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				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) SEPTEMBER 2021		2. REPORT TYPE FINAL TECHNICAL REPORT		3. DATES COVERED (From - To) NOV 2015 – MAR 2021	
4. TITLE AND SUBTITLE INTERFACES, MODELS, AND MONITORING FOR RESOURCE-AWARE TRANSFORMATIONS THAT AUGMENT THE LIFECYCLE OF SYSTEMS (IMMORTALS)				5a. CONTRACT NUMBER FA8750-16-C-0044	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 61101E	
6. AUTHOR(S) Peter Samouelian Partha Pal Austin Wellman Jacob Staples Eric Walkingshaw Arash Termehchy Alex Groce Chris Thompson Heng Yin				5d. PROJECT NUMBER BRAS	
				5e. TASK NUMBER SB	
				5f. WORK UNIT NUMBER BN	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Raytheon BBN Technologies Corp. 10 Moulton Street Cambridge MA 02138				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA DARPA 525 Brooks Road 675 North Randolph Street Rome NY 13441-4505 Arlington VA 22203-2114				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI	
				11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2021-164	
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. PA# AFRL-2021-3185 Date Cleared: 21 September 2021					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT The Interfaces, Models, and Monitoring for Resource-aware Transformations that Augment the Lifecycle of Systems (IMMoRTALS) project developed an ensemble of tools to (1) discover software metadata related to its dependence on the ecosystem, (2) process evolutionary pressure in the ecosystem, and (3) adapt the software so that it can function in the new environment and satisfy the changed requirements. The technologies were demonstrated in the context of eight challenge problems over the span of the three phases of the project, and evaluated by independent evaluation teams.					
15. SUBJECT TERMS Software Engineering, Software Adaptation, Discovery and Analytics of Software Ecosystem					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 189	19a. NAME OF RESPONSIBLE PERSON STEVEN L. DRAGER
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) N/A

TABLE OF CONTENTS

Section	Page
LIST OF FIGURES	iv
LIST OF TABLES	vii
1.0 SUMMARY.....	1
2.0 INTRODUCTION	3
2.1 Project Objective	3
2.2 Project Overview	3
2.3 Summary Accomplishments.....	4
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES	7
3.1 Background.....	7
3.2 Assumptions	8
3.3 Approach Taken.....	8
4.0 RESULTS AND DISCUSSION.....	10
4.1 DAS Overview.....	11
4.1.1 Knowledge Representation (KR) and Supporting Tools.....	13
4.1.1.1 KR BACKGROUND.....	13
4.1.1.2 IMMORTALS ONTOLOGY RATIONALE	14
4.1.1.3 IMMORTALS ONTOLOGY STRUCTURE.....	15
4.1.1.4 ONTOLOGY TOOLCHAIN	20
4.1.2 Generic Program Analysis Based KBs.....	24
4.1.2.1 OVERALL OBJECTIVE.....	24
4.1.2.2 KEY TECHNIQUES	25
4.1.2.3 SPECIFIC ANALYSES.....	26
4.1.3 Application ingestion, CM and VM.....	29
4.1.3.1 DAS INITIALIZATION AND APPLICATION INGESTION	29
4.1.3.2 ADAPTATION STEPS THROUGH CHANGE MANAGER AND VALIDATION	30
4.2 Handling Changes in Requirements	32
4.2.1 Component (DFU) Substitution (Phase 1, CP1)	32
4.2.1.1 MOTIVATION.....	32
4.2.1.2 CP DESCRIPTION.....	32
4.2.1.3 RELEVANT IMMORTALS MECHANISMS AND CAPABILITIES	33
4.2.1.4 EVALUATION.....	42
4.2.2 Handling New Requirement on Client-Server Dataflow	46
4.2.2.1 MOTIVATION	46
4.2.2.2 CP DESCRIPTION.....	46
4.2.2.3 RELEVANT IMMORTALS MECHANISMS AND CAPABILITIES	47

4.2.2.4 EVALUATION.....	59
4.3 Handling Changes in Deployment Environment.....	62
4.3.1 Adapting to Resource Constrained Deployment Environment (Phase 1, CP2)	62
4.3.1.1 MOTIVATION.....	62
4.3.1.2 CP DESCRIPTION.....	62
4.3.1.3 RELEVANT IMMORTALS MECHANISM AND CAPABILITIES	63
4.3.1.4 EVALUATION.....	68
4.3.2 Adapting Flight Test Plans (Phase 3, CP1).....	71
4.3.2.1 MOTIVATION.....	71
4.3.2.2 CP DESCRIPTION.....	71
4.3.2.3 RELEVANT IMMORTALS MECHANISMS AND CAPABILITIES	73
4.3.2.4 EVALUATION.....	84
4.4 Handling Changes in Data Representation and Schema.....	87
4.4.1 Updating Database Application to Schema Updates (Phase 2, CP1).....	87
4.4.1.1 MOTIVATION.....	87
4.4.1.2 CP DESCRIPTION.....	87
4.4.1.3 RELEVANT IMMORTALS MECHANISMS AND CAPABILITIES	88
4.4.1.4 EVALUATION.....	97
4.4.2 Updating Applications to MDL Schema Changes (Phase 3, CP2)	101
4.4.2.1 MOTIVATION.....	101
4.4.2.2 CP DESCRIPTION.....	101
4.4.2.3 RELEVANT IMMORTALS MECHANISMS AND CAPABILITIES	102
4.4.2.4 EVALUATION.....	110
4.5 Handling Changes in Software Stack	116
4.5.1 Adapting to Changes in 3 rd -Party Libraries (Phase 2, CP3).....	116
4.5.1.1 TRANSFORMING ANDROID CODE USING PROGRAM QUERY LANGUAGE (PQL)	117
4.5.1.2 HIERARCHICAL DELTA DEBUGGING BASED RESOURCE ADAPTATION	126
4.5.1.3 SELECTIVE LIBRARY UPDATE.....	131
4.5.2 Adapting ROS1 Code to ROS2 - a Machine Translation Approach (Phase 3, CP 3).....	136
4.5.2.1 MOTIVATION.....	136
4.5.2.2 RELEVANT IMMORTALS MECHANISMS AND CAPABILITIES	136
4.5.2.3 EVALUATION.....	140
5.0 CONCLUSIONS.....	144
6.0 RECOMMENDATIONS	146
7.0 REFERENCES	148
APPENDIX A - DELIVERABLES AND PUBLICATIONS	152
APPENDIX B- ADDITIONAL EXPLORATION ML-BASED CODE TRANSLATION.....	155
APPENDIX C- EXTENDING AND APPLYING PQL IN A TRANSITION CONTEXT.....	164
LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS	178

LIST OF FIGURES

Figure	Page
Figure 1: Summary of IMMoRTALS R&D over phases, technology and evolutionary pressure.....	10
Figure 2: Discovery and analysis in IMMoRTALS	10
Figure 3: DAS configuration for schema evolution	11
Figure 4: Knowledge repository toolchain components in action	23
Figure 5: Organization of resource usage and dependency discovery framework.....	25
Figure 6: Memory allocation tracking	28
Figure 7: Application ingestion process.....	29
Figure 8: Adaptation workflow	30
Figure 9: Variation Point Example	35
Figure 10: Usage Pattern-Call Site Mapping	36
Figure 11: DFU Metadata	37
Figure 12: DFU Annotations.....	37
Figure 13: Deployment Model	38
Figure 14: Mission Specification.....	39
Figure 15: Metric Definition	39
Figure 16: DFU Search Query	40
Figure 17: CP 1 Intent Outcome.....	44
Figure 18: Illustration of Reach Factor driven hybrid analysis	49
Figure 19: Intra-method.....	50
Figure 20: Intre- and Inter-network	51
Figure 21: Inter-method	51
Figure 22: Illustration of data frame analysis through compressor and encryption DFUs..	52
Figure 23: Dataflow (top) and actual code indicating the object and invocation to be proxied	55
Figure 24: Code and schematic representation of inserting the proxy	56
Figure 25: Regression tests passed per test group for Phase 2 CP2	60
Figure 26: Evaluation summary for CP2.....	61
Figure 27: Property Action Chain for Bandwidth Reduction	64
Figure 28: ConsumingPipe Interface	66
Figure 29: Illustration of DFU Composition	66
Figure 30: Schematic of the BitmapReader DFU	67
Figure 31: Schematic of the composed DFU.....	67
Figure 32: Simplified code listing of the composed DFU	68

Figure 33: CP 2 Outcomes	69
Figure 34: CP 2 Verdict Outcome	70
Figure 35: Sample flight test configuration	71
Figure 36: Raw Simplified FTC Device Excitation Definition.....	75
Figure 37: Aligned FTC Device Excitation Definition.....	76
Figure 38: Raw Simplified DAU Port Excitation Definition.....	77
Figure 39: Aligned DAU Port Excitation Definition.....	78
Figure 40: Transformed Flight Test Configuration Measurements	79
Figure 41: Transformed DAU Inventory Measurements.....	79
Figure 42: MDL Flight Test Plan adaptation evaluation workflow.....	84
Figure 43: Execution time as a function of configuration options.....	85
Figure 44: Baseline CoT Schema.....	87
Figure 45: Perturbed Schema	88
Figure 46: Sample Horn Definition Containing One Clause	89
Figure 47: Sample Conjunctive Query in SQL	89
Figure 48: Castor's cover-set algorithm	90
Figure 49: SQL Dependency Metadata.....	92
Figure 50: Sample Parameterized Query	93
Figure 51: Rewrite of Parameterized Query	94
Figure 52: Example of Semantically Inequivalent Queries.....	99
Figure 53: Schema Evolution (MIT Evaluation).....	100
Figure 54: Exemplar client-server application used in this CP	101
Figure 55: Annotation specifying the schema of the XML document argument.....	103
Figure 56: Trace of early stages of a SOAP/XML interaction with a remote endpoint.....	104
Figure 57: Hypothetical trace linking an application entry point to a network I/O cutpoint	105
Figure 58: Actual traces connecting a clientAction() entypoint to OutputStream.write()	105
Figure 59: Schema mapping and translation process.....	106
Figure 60: Output of the comparison step.....	107
Figure 61: The mapping process	108
Figure 62: Bytecode diff relative to modified classpath element (L) and modified element (R)	111
Figure 63: Idealized improvement in performance after adaptation (human-in-the-loop)	111
Figure 64: Improvement in performance using fully automated schema translation.....	112
Figure 65: Change in number of bad elements across MDL translation adaptation space	113
Figure 66: Change in number of bad lines across MDL translation space	114

Figure 67: Change in fraction of problematic documents across MDL translation space	.114
Figure 68: Bytecode modification statistics in SwRI evaluation	116
Figure 69: Android API 23 Migration	118
Figure 70: PQL Toolchain Architecture	119
Figure 71: Structure of PQL Program	120
Figure 72: PQL referenceSearch Example	120
Figure 73: Simplified PQL IDE	121
Figure 74: PQL Internal Class Diagram	122
Figure 75: PQL 'then' Section	123
Figure 76: Third-Party Library Evolution Tests	125
Figure 77: Schematic depiction of test based minimization realized by hddRA	128
Figure 78: Architecture overview of LIBBANDAID for performing selective library update	131
Figure 79: Value-sensitive Differential Slicing algorithm	133
Figure 80: Results of experiments on nine Android TPLs	134
Figure 81: Framework of learning based approach for migrating ROS1 code to ROS2 code	136
Figure 82: Calculation process of LSA for ROS migration prediction	137
Figure 83: Network architecture of the Sequence-to-Sequence model	138
Figure 84: Performances of different classifiers for ROS migration prediction	140
Figure 85: Evaluations of hyper-parameter tuning of the Sequence-to-Sequence model	...141
Figure B-1: JSON sample API	 157
Figure B-2: Sample JSON code for createFlattened API	 157
Figure B-3: Lexicon for JSON samples	 160
Figure C-1: Usage pattern annotation of a Location Provider implementation	 166
Figure C-2: LSTM model applied to migrate ROS-1 code to ROS-2	 167
Figure C-3: Illustrative example of a PQL query and transformation in DSL code	 171
Figure C-4: A portion of the Grammar for PQL's query and transform DSL	 172
Figure C-5: The PQL toolchain	 173
Figure C-6: PQL IDE showing code diff and change review options	 174

LIST OF TABLES

Table	Page
Table 1: List of software metadata that can be discovered by program analysis.....	24
Table 2: Resource Indicators (pre-defined).....	26
Table 3: Resource Factors.....	32
Table 4: Location Provider DFUs	33
Table 5: IMMoRTALS concepts and mechanisms relevant to Phase 1 CP1	34
Table 6: Evaluation Tests	43
Table 7: Test Stages	44
Table 8: Default Perturbation Profile	44
Table 9: Sample CP configuration	59
Table 10: DFU metadata example	63
Table 11: CP 2 Perturbation Parameters	69
Table 12: CP 2 Intent Verdict.....	69
Table 13: MDL to DSL transformation sequence.....	74
Table 14: FTC and DAU requirements with different transformation configurations	78
Table 15: Metrics collected from SwRI evaluation.....	86
Table 16: Data DFUs for Evaluation.....	97
Table 17: Schema Evolution Trials	98
Table 18: SwRI evaluation results.....	115
Table 19: Expected XML validation errors.....	115
Table 20: Efficiency gained by heuristics (as %) broken down by omitted test percentage for H3	129
Table 21: Heuristic Accuracy	129
Table 22: Percent improvement achieved by ADFL	129
Table 23: Statistics on the generated ROS1 and ROS2 migration dataset.....	137
Table 24: Hyper-parameter configurations evaluated in experiments.....	141
Table 25: ROS conversion results	143
Table B-1: Studied code migrations.....	156
Table B-2: PQL results summary.....	163

1.0 SUMMARY

Interfaces, Models, and Monitoring for Resource-aware Transformations that Augment the Lifecycle of Systems (IMMoRTALS) is a project under the Defense Advanced Research Project Agency (DARPA) Building Resource Adaptive Software System (BRASS) Program. The BBN-led team consists of Securboron, Oregon State University (OSU), Vanderbilt University, and University of California Riverside (UCR). The project is sponsored by the Air Force Research Laboratory (AFRL) under the Contract No. FA8750-16-C-0044.

The goal of the project was to develop and demonstrate technologies and tools that enable software applications to adapt and evolve over time in response to changes in its ecosystem, including changes in the resources available in the application's operating environment as well as changes in the requirements of the missions it serves. BRASS recognized the difficulty of addressing the problem in its full generality, and called for *platforms* in which software adaptation technologies and evolution can be developed and demonstrated. Initially, IMMoRTALS focused on a tactical situational awareness information sharing platform. The software in this platform consists of an Android client application and a Linux publish-subscribe (pub-sub henceforth) server. In a given deployment, multiple clients connect to and interact with each other via the pub-sub server. In Phase 3, the focus of the project shifted to technology transition, and relevant IMMoRTALS capabilities were applied and enhanced in the context and change drivers of transition partners like Southwest Research Institute (SwRI) and the Army Tank Automotive Research, Development and Engineering Center (TARDEC).

Over the course of the three phases of the project, the IMMoRTALS team considered four main classes of change drivers: changes in requirements of the application, changes in the deployment environment, changes in the data representation, and changes in the software stack that the application depends on. The Research and Development (R&D) hypothesis underlying the IMMoRTALS technical approach that crosscuts these diverse change drivers is as follows: *is it possible to encode metadata based recipe such that given appropriate building blocks, the recipe can respond to a given change by automatically adapting the software*. In contrast to writing a program to effect a specific change, this approach requires additional knowledge about the application (software metadata), encodes logic to identify where changes are needed, search for the appropriate changes, and make them. When building blocks with appropriate metadata are available, the encoded logic—the recipe – can automatically adapt the application in response to a given change driver.

To realize this concept as an augmentation to existing application build tool chain, the IMMoRTALS team developed a number of discovery capabilities to extract the knowledge—software metadata, and analysis capabilities to effect the adaptation recipes corresponding to specific evolutionary pressures. The discovery and analysis capabilities are integrated in the IMMoRTALS Discovery and Analysis System (DAS) that also includes interfaces for change detection or submission, integration with compile toolchain for building executables, and support for testing the modified software application.

The R&D methodology the IMMoRTALS team followed is based on rapid prototyping. Once a change driver or evolutionary pressure is identified, the knowledge building discovery capabilities as well as the adaptation modules responsible for the analysis and code modification were built

incrementally, and in many cases in parallel. The validation of the developed solution for the given change driver was performed in the context of a challenge problem, first within the team, and then by the independent evaluation team. For transition, the IMMoRTALS team also developed lightweight DAS configurations where a streamlined pipeline combined the discovery (knowledge builder) and analysis (adaptation module) capabilities without the need for the heavier weight ontology-based common knowledge representation and simply assumed the relevant knowledge and internal data representation is maintained by the streamlined pipeline in the form that is internal to itself but also best suited for its own purpose. The learning-based pipeline to migrate Robot Operating System version 1 (ROS1) code to Robot Operating System version 2 (ROS2) is an example of this approach. This enables faster adoption and turnkey use of the IMMoRTALS technology, but lacks the advantage of being combined with the other discovery and analysis capabilities of the IMMoRTALS DAS.

Apart from establishing the feasibility of the overall approach, IMMoRTALS also advanced the state of the art in associating programmer defined knowledge to software based on a pre-defined vocabulary and extracting that metadata into a knowledge store for further processing, discovery capabilities that extract metadata about the software (knowledge builders), analysis capabilities that help determine what changes are needed where in the program to accommodate a given ecosystem change, and lightweight integrated discovery and analysis pipelines that address specific types of change drivers. The accomplishments of the IMMoRTALS project along with technical publications and open source software releases are described in Sections 3 and 4.

Overall, the IMMoRTALS effort, which is a fundamental research (6.1) project, largely achieved its stated goals of developing and validating new technologies that facilitate adapting software in response to ecosystem changes. In addition, the project demonstrated the utility of developed technology in the context of the Department of Defense (DoD) transition partners. However, the reason we caveated the success with *largely achieved* is that more work remains to be done. In particular, currently the amount of knowledge used (and related to that, the amount of human effort needed to create, collect or extract that knowledge) to address realistic anticipated and unanticipated ecosystem changes that impact the software lifecycle in a non-trivial manner, are considerable. It is unclear at this point, whether it is possible to define a necessary and sufficient bound on the information needed to address the changes the software might encounter, and whether all such information need to be collected up front, or optimization or on-demand mechanisms are feasible. Second, a thorough evaluation quantifying the net reduction in the amount of human effort needed to respond to ecosystem changes with IMMoRTALS (which will involve figuring out the right metrics for determining human effort expended in adapting software and how to measure them with and without IMMoRTALS technologies) is also needed for convincing the potential users of IMMoRTALS technologies and early adopters. At the same time, the success we observed with the lightweight mechanisms to address narrow and specific change drivers makes us ask the following question: what would be a better way to transition emerging technologies like IMMoRTALS? Paradigm shift- where the community has to buy into the vision of metadata and knowledge driven software development and adaptation, or incremental augmentation- where lightweight tools customized for a specific software can help developers address specific ecosystem changes for that software? More work is definitely needed to disambiguate this dilemma.

2.0 INTRODUCTION

2.1 Project Objective

The objectives of the IMMoRTALS project can be summarized as follows:

- **Develop an information model and supporting toolchain:** Since the IMMoRTALS approach is based on knowledge and information about the software itself, a model of how the software metadata is captured in a machine digestible form and tools supporting the use of the model in adding, extracting and effectively accessing software metadata are needed.
- **Develop discovery and knowledge extraction capabilities:** Even though IMMoRTALS allows developers to add metadata (knowledge) to code, it is expected that only a small portion of the knowledge needed for automated adaptation will be explicitly added by the developers. The rest need to be automatically discovered. Therefore, one of the main objectives was to develop tools and capabilities to extract dependency, resource usage and other salient information about software.
- **Develop analysis and adaptation modules:** Since IMMoRTALS focused on modifying code within the larger space of BRASS's *resource adaptive software systems*, automated code modification capability is a key aspect of IMMoRTALS. Instead of trying to develop a universal code modification engine, we set out to build adaptation building blocks—pluggable analytical modules that can perform specific code modification tasks.
- **Develop and demonstrate an integrated DAS:** This objective aimed to demonstrate integrated operation of knowledge building and adaptation capabilities interoperating over the triple store in a standalone DAS. The DAS can respond to ecosystem changes by adapting code and producing a new binary that passes the applicable tests, or provides the details of the explored search space for the human developers to take over.
- **Explore technology transition:** Like any other R&D project developing new capabilities, IMMoRTALS also had a goal to explore practical use of the newly developed technology. It is rarely the case that a new capability goes directly from the R&D and proof of concept to production-grade usage. IMMoRTALS had a modest goal to getting its technologies to a readiness level that is appropriate for early adopters to evaluate the new capabilities in their own context.
- **Human development:** IMMoRTALS aimed to raise awareness of the brittleness of software against ecosystem changes, emerging potential solutions, and approaches to build better maintainable, resource adaptive software systems among the next generation of practitioners.
- **Knowledge dissemination:** This objective involved sharing of research findings, both positive and negative results, through technical presentations, and publications in conferences and journals.

2.2 Project Overview

Consistent with the structure of the BRASS program, the IMMoRTALS project was organized in three 16 month phases. Challenge Problems (CPs) set the technology development, demonstration and evaluation target in each phase. Phase 1 has two challenge problems, and Phases 2 and 3 had

3 each. Solving, or developing the solution of an individual CP, involved rapid prototyping style spirals.

As noted earlier, IMMoRTALS focused on addressing ecosystem changes through code modification. We also wanted to explore a smallish part of the change drivers/adaptation space in some depth. Therefore, the CPs we chose covered changes in the following general areas, with different CPs highlighting related problems and synergistic discovery and analysis capabilities within each:

- Changes in requirements: the application needs to satisfy new requirement.
- Changes in deployment environment: the application needs to operate in a different environment with different constraints on resource availability.
- Changes in data representation: the representation format of the 3rd party data that the application uses has been changed by the vendor.
- Changes in software stack that the application is dependent upon: adapt to cope with the evolution of the OS, framework, or 3rd party libraries when such changes are not backward compatible.

IMMoRTALS and the BRASS program itself are early explorers of the *automated adaptation in response to ecosystem changes* problem space. Therefore, the goal was to develop proof of concept technologies and evaluate, as opposed to a complete/exhaustive exploration of the space. As a matter of fact, based on our experience, we argue that completeness is probably unachievable since the problem space itself seems unbounded- as the technology landscape evolves and new applications emerge, even the classes of change drivers or evolutionary pressures seem open ended.

2.3 Summary Accomplishments

The achievements of the IMMoRTALS effort can be summarized as follows:

- Established that the software metadata driven approach to accommodate ecosystem changes through code modification is viable, although it requires a lot of metadata. But the biggest success is perhaps in the focused, lighter weight application of specific combination of knowledge builders and adaptation modules to address specific change drivers in a given application context.
- Developed knowledge representation model and vocabulary, and supporting tools. Creating a vocabulary backed by an ontology for knowledge representation often suffer from scope creep (modelers tend to include more than what is needed in the model). To avoid that risk, we deliberately followed an on-demand approach- we started with a small and reasonable base of components and relations needed to describe software, and then extended that model as needed. As a result, our knowledge representation is somewhat ad-hoc and incomplete, and may need extension to handle new software or new change drivers. We used the Apache Jena framework for representing linked data in RDF triples, and Fuseki for serving Resource Description Framework (RDF) triples over Hypertext Transfer Protocol (HTTP). Among other tools, we created a Java annotations based toolchain to allow adding ontology/knowledge model-based metadata to code, and extractions of RDF triples from the annotated code.

- Developed the following discovery capabilities (knowledge builders) responsible for extracting software metadata:
 - Program analysis: static and dynamic analysis of Java and C++ code to discover resource dependency, resource usage as well as dependency among software component through calls, data and control flows.
 - Byte code analysis: byte code analysis to identify control and data flow dependency among architectural components, as well as handoff points between data handling components and layers (e.g., system libraries wrapping hardware resources such as memory and network).
- Developed the following analysis capabilities (adaptation modules)
 - Source and Bytecode transformation: transforms source code to use the new learned relational query; transforms bytecode to splice in new data handlers.
 - Substitute and compose discrete functional code units in a Product Line Architecture (PLA): substitutes a code unit by another that satisfies the new requirements, resource profile or dependencies, taking care of existing differences in their constructor, use and destructor functions; creates a new code unit by invoking another code unit from within one, effectively composing the functionalities of the two components.
 - Test driven minimization: eliminates code that is not essential for a given context defined by a set of tests that must pass, and another optional set of tests that are treated as wild cards in the present context. *Note- although the effective result of this tool is code deletion, the technique can be described as a discovery-analysis pipeline where running the tests can be thought of a discovering knowledge and dependencies.*
 - Schema mapping: infer the mapping (unchanged elements, newly added elements, deleted elements, renamed and/or moved elements etc.) between two given versions of a document descriptor schemas (e.g., eXtensible Markup Language or XML schema). More details in Section 4.4.2. Additional source or bytecode modification is applied to make use of the learned mapping in the adapted code.
 - Relational learning: given the positive and negative results of a query Q against a data set D represented in a relational schema S1, and the same data set D represent in a new schema S1, learn the new version of the query Q that would produce the same results against D represented in S2, as long as the difference between S1 and S2 follows certain conditions. More details in Section 4.4.1. Here also, additional source code modification is performed to use the learned query in the adapted code.
 - Resource Domain Specific Language (DSL): treats resource requirements of PLA components as types and performs type checking (i.e., checking if the environment satisfies the resource requirements of a given component), type resolution (i.e., finds a component that fits within the available resources in the current environment) and optimization (i.e., finds the component that optimizes a utility function with respect to the current resource availability). Source and bytecode transformation techniques are used after such resource type checking, resolution and optimization to use the validated/selected component in the code.

- Developed lightweight DAS discovery-analysis pipelines tailored to address specific evolutionary pressures in the context of a given application:
 - Machine learning based ROS migration: achieved over 90% correctness to identify which statements in a given ROS1 code (i.e., code conforming to ROS1 framework) will need to be transformed to make it compatible with ROS2, and achieved ~70% correctness in applying the required transformations to the identified candidates.
 - Selective update: identifies self-complete code segments that include security patches in a library version, and integrates that code segment in an older version of the library so that users of the older version of the library can take advantage of the security patch without having to switch to the new version. This selective update is extremely helpful in the fairly common situation in the software community where a confluence of various dependencies make upgrading to a newer version of a library breaks many other dependencies and requires a lot of code changes.
 - Program Query Language (PQL): a domain specific language and supporting compiler to transform source code based on specified patterns. This tool is handy in automating large volume of existing code, given an expert developer or another tool (e.g., a machine learning based tool) provides the patterns specifying the transformation.

We demonstrated the IMMoRTALS capabilities in the context of the following transition partners:

- TARDEC: Migration of ROS1 code to ROS2 using machine learning based discovery-analysis pipeline.
- SwRI: Transformation of Metadata Description Language (MDL)-based test plan to accommodate a failed test equipment such as a data acquisition unit (DAU), and adapting Java code that ingests MDL documents when it needs to handle documents conforming to a new MDL version.
- AFRL and ROGER/TAKServer community: Use of PQL to update TAKServer and ROGER plugin modules when TAKServer and ROGER frameworks are updated.

In terms dissemination of the research results, the IMMoRTALS's accomplishments include:

- Papers published: over 20 papers published in international conferences and reputed journals, and a few more are in the submission pipeline. (See Appendix A for the list)
- Open source software: IMMoRTALS developed capabilities that are available as open source software (OSS) are listed in Appendix A.

The IMMoRTALS project supported the following graduate students and academic research staff:

- University of California Riverside (UCR): Yue Duan (Graduated with a PhD in 2018). In addition, PhD students Sri Shaila, Lian Gao and Jie Hu were partially supported. Prof. Yin's Post Doc Yu Qu and visiting PhD student Jianlei Chi were also supported.
- Oregon State University (OSU): Arpit Christy (Graduated with a PhD in 2019) and Jose Picado (PhD candidate). In addition Spencer Hubbard (MS 2016, was supported for 1 year), Keeley Abbott (MS 2017, was supported for 1 year), Alexander Grasley (MS 2018, was supported for 1 year), Parisa Ataei, (PhD candidate, was supported for 2 years), and Jeffrey Young (PhD candidate, was supported for 3 years)

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

In this section we describe the background that motivated the IMMoRTALS solution, the underlying assumptions and starting points for the R&D we conducted, as well as the methods we followed in developing and evaluating the IMMoRTALS solution.

3.1 Background

Software applications today are developed, deployed and operated in the context of a complex ecosystem consisting of hardware, various layers of software, as well as external services. Known and unknown dependencies exist between the software application and the ecosystem elements, some of which may be well studied whereas others are much less understood beyond their existence. As a result, performance of the software application, including its correct functioning- is not only a matter of having code that implements the specification and compiles- a lot depends on the assumptions the code makes about its ecosystem, and whether hardware and software resources in the ecosystem satisfy those assumptions.

Computer systems and software technology landscape is evolving at a fast pace. New and more powerful hardware, and more abundant hardware resources appear in the marketplace at regular and ever-shrinking intervals. Operating System (OS) and software vendors churn out new versions, bug fixes and enhancements at an even faster rate. At the same time, software applications are used and deployed in newer contexts with new and different requirements from what they were originally intended for. Any such change in the ecosystem and deployment environment, or the requirement to support a new operating point has the potential to break the intricate dependencies between the application code and its ecosystem, causing the application to degrade or stop functioning at all. When that happens, the state of practice is to manually modify the software which often involves significant amount of redesigning, reimplementation and retesting. In rare occasions, if the application is designed and built appropriately, it may be possible to dynamically switch operating points, change configuration settings or even download new code (e.g., dynamic class loading). But for the most part, especially in the DoD context, ecosystem changes limit the shelf life of software applications, because handling such changes involves a new release of the application requiring the system owners and users to follow the procurement and software upgrade processes. This expensive and time consuming approach is untenable in today's fast paced and highly dynamic warfighting environments.

The BRASS program aims to extend the shelf life of software products, by developing technologies that will automatically evolve software applications without having to go through the procurement and upgrade processes to replace the current software with a manually constructed new version. As part of the BRASS program, the IMMoRTALS project mainly focused on programmatically changing application code and producing a new executable that accommodates the ecosystem change. We used a tactical situational awareness information management system as a platform to demonstrate and evaluate the IMMoRTALS technical capabilities. The IMMoRTALS capabilities are envisioned to augment the current software development lifecycle with automated discovery and analysis steps that can be performed by the developers of the application as well as by the system owners who are responsible for its deployment and maintenance, IMMoRTALS did not focus on runtime adaptation/reconfiguration of a running application.

3.2 Assumptions

We make the following assumptions in order for IMMoRTALS technologies to work correctly:

- Availability of application build tools: Build tools to compile the application software and create application binaries are available for use by IMMoRTALS technical capabilities.
- Availability of tests: Correctness of the software evolved by IMMoRTALS is assessed by passing tests- therefore sufficient tests along with test inputs and success criteria, covering key requirements must be available.
- Availability of software building block: software components that are not necessarily part of the application, but performing various relevant, general purpose and ancillary computation (e.g., encryption or compression, various image, string, or file manipulation) annotated with their functional descriptions are available for use by IMMoRTALS. IMMoRTALS capabilities do not create new code from scratch, rather they mine these available building blocks for code fragments to substitute or replace application code or combine (compose) with application code.
- Application architecture: The application which is being evolved using IMMoRTALS must follow a well-defined architecture. The application architecture provides a structure that the evolving application must maintain, and therefore acts as guiderails for discovery and analysis techniques that manipulate application code.
- Information model: IMMoRTALS relies on knowledge about software. This knowledge needs to be in machine analyzable form. In order for that to happen, an information model- a knowledge representation formalism consisting of a vocabulary and ontology is needed. We assume that this model will be available for the application to which IMMoRTALS is applied. In the current project, we built the vocabulary and ontology ourselves. IMMoRTALS capabilities include an ontology and annotation tool chain that enables programmers to annotate Java code with meta-information using a given ontology and an annotation-processor to extract the knowledge embedded in code annotations to triples that can be queried and processed algorithmically.

Another underlying assumption in IMMoRTALS is that human involvement will be needed to habilitate the auto-modified programs into the source code repository. Auto-generated code is not very readable. If the evolved code is envisioned to be used beyond the immediate need (i.e. handling the current change drivers), it needs to be vetted and cleaned by human experts before they added back to the maintained repository in the appropriate source branch.

3.3 Approach Taken

The IMMoRTALS team followed a rapid prototyping approach consisting of multiple cycles of design- implement-evaluate-feedback steps toward developing knowledge builder and adaptation capabilities needed to execute individual challenge problems. At any given time within a program phase, the team worked on multiple challenge problems in parallel.

Over the phases, the challenge problems were carefully chosen to focus on a handful of change drivers (see Section 3.2) so that the discovery (knowledge builder) and analysis (adaptation)

techniques developed have some synergy among themselves, and get an opportunity to be refined over time, as opposed to branching into new capabilities for each challenge problem.

The design step started with selecting and defining the challenge problem and ended with a structured description of the challenge problem to be reviewed by the independent evaluators, as well as the requirements for the discovery and analysis capabilities needed to support the challenge problem.

The implementation step involved developing or extending knowledge building and adaptation capabilities, and unit testing them. This was followed by integrating them together so that code modification can use the metadata derived by the knowledge builder, triggered by changes provided as inputs. Once this discovery and analysis integration stabilized—i.e., reliably produced code modifications—build tool and testing was added to the sequence such that failure in compilation or testing forces the toolchain to explore alternate adaptations. Knowledge building and adaptation capabilities often relied upon externally developed techniques such as various static and dynamic program analysis and machine learning techniques to build upon.

For evaluation, we first had to move the working solution of the challenge problem from IMMoRTALS development environment and install it in the independent evaluation team's evaluation harness. This process was repeated in multiple risk reduction milestones. After preliminary testing with our homegrown test data/inputs, the final version of the solution was handed off to the evaluation team, which tested and evaluated the delivered solution using their own data and test inputs.

The evaluation team provided informal feedback and bug/issue reports during the risk reduction milestones, which provided the IMMoRTALS team a window to address the identified issues and shortcomings until the code freeze milestone. The final evaluation report was provided to the BRASS community at the end of phase workshop.

4.0 RESULTS AND DISCUSSION

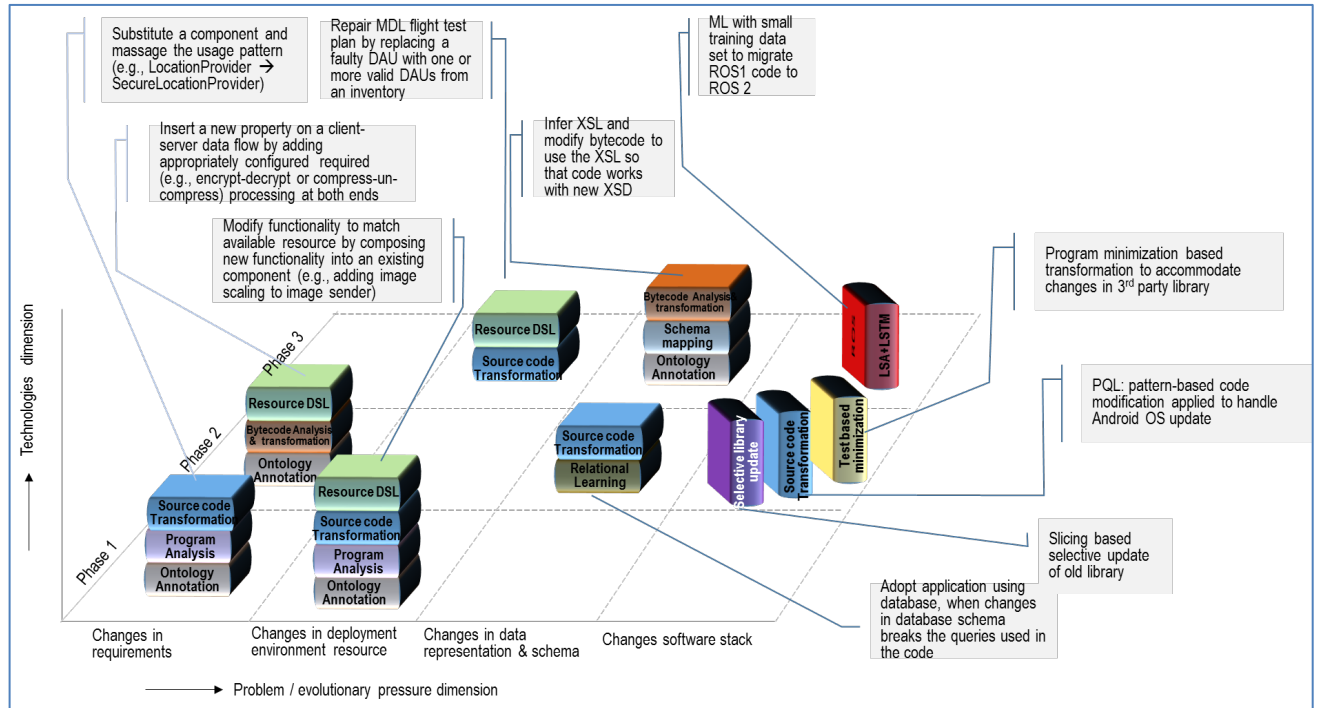


Figure 1: Summary of IMMoRTALS R&D over phases, technology and evolutionary pressure

Figure 1 summarizes the IMMoRTALS R&D results. The X axis shows the various evolutionary pressures IMMoRTALS technology handles, the Y axis shows what was accomplished in different phases and the Z axis shows the various technology components that individually or collectively addressed specific evolutionary pressure.

As mentioned earlier, the IMMoRTALS team focused on lightweight adaptation tools that can be easily applied in a small number of steps in the later part of the project (see the vertical tiles or the stacks that do not have the ontology tile in Figure 1).

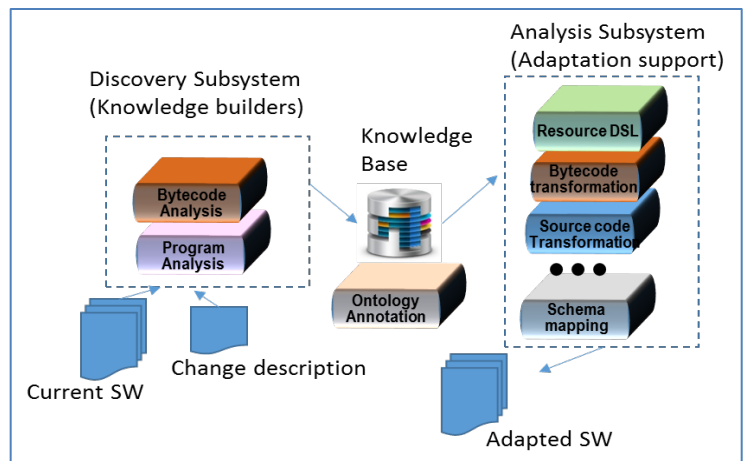


Figure 2: Discovery and analysis in IMMoRTALS

Figure 2 shows how the discovery techniques feed the knowledge base and the analysis techniques use the knowledge base to adapt the candidate software for a given change description. The change description can be in the form of new requirements, modified deployment environment, data describing change (e.g., new schema, or the positive and negative results of a query to be learned against a database with updated schema). The rest of this section describes the R&D findings organized in terms of the evolutionary pressures addressed, along with the challenge problem used to demonstrate and evaluate the capability.

4.1 DAS Overview

As noted earlier, the IMMORTALS technology portfolio consists of multiple discovery/learning and analytic/adaptation modules. Analyses embodied in an adaptation module may be tailored to address a specific type of evolutionary pressure and even depend on outputs from specific knowledge builders, yet all knowledge builders and adaptation modules rely on a common architecture, and information model and a set of auxiliary components. This common functionality and architecture, together with the custom adaptation modules, make up the Immortals Discovery and Analytic Services (DAS) system. The DAS architecture defines the following components:

- Knowledge Builders (KBs). KBs are components that perform source and byte code analysis including static and dynamic analyses to collect and store information for later use when adaptive triggers occur. The DAS system executes KBs during the build cycle of the target application, and stores the discovered software metadata (knowledge) in a central RDF data store,
 - Except for the transition focused CPs, our target application was the tactical situational awareness information management system, consisting of the TAK (Team Awareness Kit) Server and ATAKLite applications. ATAKLite is an Android application that is a lightweight version of ATAK, the Android Team Awareness Kit. The application transmit and receive location reports to and from TakServer, which acts as a central hub to receive and route location reports, including images from the Android device cameras. Location reports are represented using the Cursor-on-Target (CoT) XML format.
- Change Managers (CMs). CMs are notified of application change events in order to give the DAS system an opportunity to adapt the target application if necessary. These events includes changes to

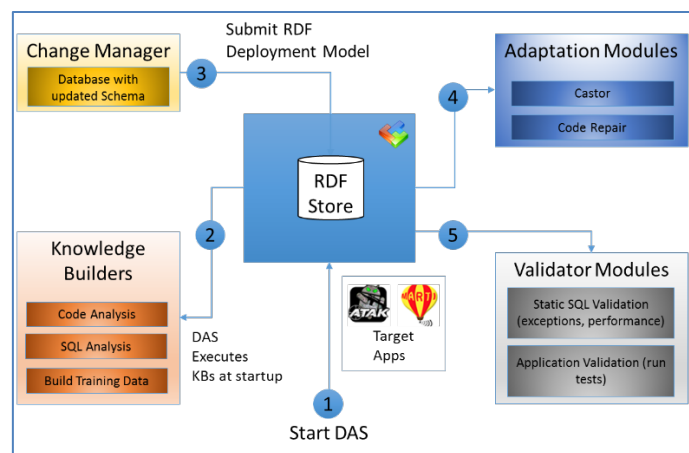


Figure 3: DAS configuration for schema evolution

dependent relational schema, changes in requirement and deployment model as well as reports from test suites.

- Adaptation Modules (AMs). These components implement various techniques that attempt to address potential problems, such as broken schema dependencies or missing resources in the application's environment.
- Validator Modules (VMs). These are simple test components that can be used to detect whether adaptation is required and, when applied, effective.

Figure 3 shows the KB and AM components used in the DAS configuration for the schema evolution CP, along with the auxiliary CM and VM components. Code and Structured Query Language (SQL) analysis KBs locate SQL dependencies within the application and analyze exactly where and how each embedded SQL string is constructed, whether the statements parameterized and which program variables or values they are assigned to, the positive and negative training data snapshots taken on the correctly functioning version of the application (i.e., the baseline application) before adaptive triggers have taken place (e.g., before the schema has been modified). When the schema evolves, the change manager is notified and uses the training data collected earlier to verify whether the SQL statements are impacted. The DAS invokes the Castor and Code Repair modules for each impacted SQL statement in order to learn a new equivalent SQL statement over the modified schema and repair the application. Finally, the VMs are executed to verify whether the repaired code and SQL statements are functioning properly.

The following types of information are stored in the central RDF repository:

- Results of static and dynamic program analysis. These are tailored for the needs of each AM. The main intent of separating knowledge building from adaptation is to give the DAS system an opportunity to capture the baseline operational state of the application before change events occur.
- The deployment model, including current mission requirements, available hardware and infrastructure resources.
- Metadata describing the system's software repository. Immortals provides a Java annotation language and toolchain to allow software engineers to provide additional information associated with their code. This includes resource dependencies, functional attributes and their usage patterns.
- Domain knowledge that is required for software adaptation.

Note that for lighter weight streamlined mode of operation, the AMs themselves perform the discovery that is necessary and do not use the information model and RDF store. This mode was leveraged for applying IMMORTALS to the CPs provided by the transition partners.

Details of the AMs handling specific evolutionary pressures, along with the supporting KBs, CMs and VMs are described in the context of the relevant CPs. The remainder of Section 4.1 describes the common aspects including the ontology, knowledge representation and generic KBs, and a general description of the CM and VM along with the DAS initialization process.

4.1.1 Knowledge Representation (KR) and Supporting Tools

In this section we describe the IMMORTALS ontology, which is used to represent the knowledge-software metadata that is needed for repair, and are extracted by the knowledge builders. The ontology serves as a bridge between semantic concepts with meaning to humans and automated repair tools that act on software.

An ontology is a conceptual model for describing things in the world (and their relationships to other things). More formally, an ontology is a descriptive logic, a structural formalism containing a subset of the expressiveness provided by first-order predicate calculus. This structure is defined using the Web Ontology Language (OWL).

4.1.1.1 KR Background

OWL [14] is a World Wide Web Consortium (W3C) language specification first released in 2004 based on work done on the DARPA Agent Markup Language (DAML) and Ontology Inference Layer (OIL). OWL is a descriptive logic. In the parlance of descriptive logics, OWL offers SHOIQ¹ expressivity [15]. A more powerful variant called OWL2 with SROIQ² expressivity was released by W3C in 2007. When we subsequently refer to “OWL” in this document, we actually refer to the OWL2 variant. There are two subset dialects of OWL designed to make tradeoffs between expressiveness and reasoner performance. OWL Lite (SHIF³ expressivity) is the simplest of these and is the only OWL dialect to achieve decidable consistency checking in polynomial time. OWL-DL (SHOIN⁴ expressivity) theoretically supports decidable consistency checking but no polynomial time algorithms to do so are known. Consistency checking for OWL and OWL2 is not generally decidable.

Abstractly, the IMMORTALS ontology is divided into two types of structures: ABox (assertional) and TBox (terminological) statements. TBox statements are axiomatic, providing structure. For example, [Human subclassOf Organism] is a TBox assertion. ABox statements are used to describe individuals. For example, [Bob instanceOf Human] is an ABox assertion. We will subsequently refer to the collection of TBox statements in an IMMORTALS ontology as a *vocabulary* and the ABox statements as *instances*.

Whereas the IMMORTALS vocabulary is relatively static over time—for example, new TBox statements are not emitted during the execution of a challenge problem—the instances change often and ABox statements are frequently created during challenge problem execution. In other

¹ SHOIQ expressivity defines a class of descriptive logic that supports attributive language with complex concept negation(S), role hierarchy (H), nominals (O), inverse properties (I), and qualified cardinality restriction (Q)

² SROIQ expressivity defines a class of descriptive logic that supports attributive language with complex concept negation(S), limited complex role inclusions axioms; reflexivity and irreflexivity; role disjointness (R), nominals (O), inverse properties (I), and qualified cardinality restriction (Q)

³ SHIF expressivity defines a class of descriptive logic that supports attributive language with complex concept negation(S), role hierarchy (H), inverse properties (I), and functional properties- a special case of uniqueness quantification (F)

⁴ SHOIN expressivity defines a class of descriptive logic that supports attributive language with complex concept negation(S), role hierarchy (H), nominals (O), inverse properties (I), and cardinality restriction (N)

words, the IMMORTALS vocabulary is reusable but individuals are specific to a given adaptation problem instance.

Note that although IMMORTALS ontologies may contain TBox axioms that utilize OWL2 or OWL-DL constructs, inferencing engines conforming strictly to OWL Lite are used for querying and consistency checking. This allows us to make interesting queries and get useful results in polynomial time.

OWL ontologies are often serialized into RDF [16] (an XML format, and as such quite cumbersome and limited by various restrictions imposed by the XML specification). The Terse RDF Triple Language (Turtle) is an alternative syntax to RDF for representing OWL ontologies that is not artificially limited by XML. Examples in this section will be presented as Turtle due to its improved human comprehensibility and brevity compared to RDF.

SPARQL Protocol and RDF Query Language (SPARQL) [17] is a W3C-defined language for querying graphical knowledge representations (such as the IMMORTALS ontology). Graphical knowledge is information that can be serialized into triples, where triples are 3-tuples comprising a subject, a predicate, and an object. For example, we might want to say that Bob (subject) has a friend (predicate) named Alice (object). In triples, this might look like

Bob hasFriend Alice

To make names within triples unambiguous, we use Uniform Resource Identifiers (URIs). So instead of using “Bob” in our triples, we’d instead provide a URI that uniquely identifies Bob in our vocabulary. Something like

<http://darpa.mil/IMMORTALS/ontology/r2.0.0/people#Bob-484e43eb>

SPARQL is a language for querying datasets containing a large number of these triples. For example, we could search for all individuals who are friends of Alice and also Jane like so:

```
SELECT ?interestingPerson
WHERE
{
    ?interestingPerson hasFriend Alice .
    ?interestingPerson hasFriend Jane .
}
```

During analysis, IMMORTALS makes use of complex SPARQL queries to retrieve the information it needs to adapt to a given problem instance.

4.1.1.2 IMMORTALS Ontology Rationale

The various tools developed by IMMORTALS team members must speak a common dialect. For example, the results of static analysis must be described in a manner useful to components that depend upon that analysis. Without this vocabulary normalization, Technical Area (TA) performers would be stuck performing fragile one-off transformations each time they communicate with another component. Any change to one performer’s output would have an

unforeseen cascading effect throughout the system. A reasonable engineering decision in this scenario is therefore to enforce the use of a common vocabulary.

There are a number of formats such a vocabulary could take. A valid line of inquiry is why the IMMORTALS team elected to use an ontology. We have outlined the following reasons:

- Utility for the task at hand
 - Descriptiveness. OWL is a SROIQ descriptive logic, but less expressive dialects (OWL-Lite, OWL-DL) can be used where needed.
 - Verifiable consistency. As a descriptive logic, an OWL ontology can be programmatically checked for consistency. An ontology is consistent if it contains no contradictory assertions. A consistent ontology is said to be a *model*, but owing to the massive overloading of this term, we simply refer to consistent OWL documents as an ontology rather than a model.
 - Inferencing. Ontology libraries such as Apache Jena or Pellet provide mechanisms to infer new facts from existing axioms in the ontology. For example, the `subClassOf` property is transitive so if `{A subClassOf B}` and `{B subClassOf C}` are explicit statements then we can infer `{A subClassOf C}`. New transitive properties can be defined to achieve a similar effect. Reasoners can be populated with additional customizable *rules* for inferring additional facts without relying upon existing constructs such as `transitiveProperty`.
 - Queryable. SPARQL provides a powerful mechanism for algebraically querying a repository.
- Exists within an ecosystem that provides future proofing, scalability, and support
 - Easily integrated with third-party schemas and knowledge representations
 - Used widely in industry and academia
 - Supported by mature, open-source tools. We primarily use Apache Jena and Fuseki

4.1.1.3 IMMORTALS Ontology Structure

The IMMORTALS ontology is organized into six distinct but related vocabularies- core, bytecode, Java project, challenge problem, feature model, and assertional. These are briefly described below.

4.1.1.3.1 Core

The core vocabulary contains the definition of general-purpose concepts central to the operation of IMMORTALS.

- Discrete Functional Units (DFUs). DFUs describe the discrete pieces of an application. For example, a software system might contain a message queue, a message parser, a message handler, and a client for a remote service—each of these could be represented as a DFU. DFUs may have one or more functional aspects, which model the *reusable* aspects of the DFU. For example, a Cipher DFU would have encrypt and decrypt aspects for capturing these obviously different yet related aspects of the same core functionality.
- Properties

- Algorithm property: configuration for how something is done. E.g., chaining mode, padding scheme, block size, key length.
- Data property: properties possessed by data as it moves through the system. E.g., encrypted, compressed (lossy=true).
- Ecosystem resources
 - Software (for example, the distinction between an operating system and the libraries needed to run an application).
 - Hardware, including devices (server device, desktop device, mobile device, embedded device); computing resources (memory, CPUs, auxiliary hardware); and IO (disk, file system, network).

4.1.1.3.2 Bytecode

The bytecode vocabulary provides the ability to describe the high-level structure of compiled Java software in a manner useful for analysis. Below are several examples of the terms in this vocabulary.

- AClass—the fundamental unit of bytecode is the class. Classes contain fields and methods, and methods in turn contain instructions that may reference fields or methods in other classes. The provenance of each compiled class is tracked in order to associate source files with their compiled forms.
- AMethod—methods contain instructions. Some instructions result in the access of fields or the invocation of other methods in other classes.
- BytecodeArtifact—describes a compiled chunk of bytecode. Examples include archive-level artifacts such as Java Archives (JARs), Web Application Archives (WARs), Enterprise Application Archives (EARs), Android Application Archives (AARs) or naked bytecode (classes). Each bytecode artifact is stored alongside a cryptographic hash value to account for the possibility that two classes may have the same name but different bytecode.

4.1.1.3.3 Java Project

The Java project vocabulary includes terminology for concepts related to maintaining Java software over its lifespan. This includes modeling the relationships between source code and the build process and any emitted binary artifacts, as well as mechanics for packaging and then executing the software after compilation.

- Classpath—a classpath is a list of bytecode artifacts that will be used by a Java Virtual Machine (JVM) to compile or execute an application. A Java project typically has several classpaths: one for building the software, one for testing the software, and one for executing the software.

- **JavaSourceFile**—contains an Abstract Syntax Tree (AST) representation of a .java code file. Although not currently used, this could theoretically be used in the future to modify and then reconstitute source files entirely from triples.
- **BuildScript**—a script used to build the software. For example, this can be achieved using an Ant, Maven, Gradle, or a shell script.
- **DfuInstance**—describes an instantiation of a DFU discovered in a Java project through bytecode-level analysis.

4.1.1.3.4 Challenge Problem

This vocabulary contains terminology necessary to achieve adaptation in practice for our first two challenge problems. Example concepts are:

- **DfuPointer/CodeUnitPointer**: an indirection mechanism that allows compile-time analysis tools to reference code structures in a manner consistent with runtime analysis tools. This technique is based on cryptographic hash values to account for the fact that there may be various versions of the same class available at different points in the application's lifecycle (e.g., before vs after adaptation).
- **MeasurementProfile**: binds together various observations about the performance of software during a single test run. This is necessary because different test runs may be associated with different configurations so any kind of sensitivity analysis needs to account for the differences between runs.
- **GmeInterchangeFormat**: a format useful for communicating between Generic Modeling Environment (GME) and the DAS. This is a convenience abstraction that provides a single point of definition for the resources available in the ecosystem and also the functional spec of the software.

4.1.1.3.5 Feature Model

This vocabulary contains terms related to the application's Feature Model (FM). The FM is a schematic of sorts that allows certain features to be selected or not. This vocabulary is preliminary but contains concepts such as:

- **Variation point**: a node in the feature model where a selection between one of various possible features is possible.
- **Features**: desired functionality to include in or exclude from the application.
- **Cutpoints**: describe the intersection of a feature with actual application code.
- **Resource dependencies**: describe the dependencies of a feature on the ecosystem.
- **Variant**: a variant is a set of selected features that satisfies the dependency relationships of the feature model.

4.1.1.3.6 Assertional

The assertional vocabulary provides general-purpose mechanisms for describing the state of things as they are (descriptive assertions), as they should be (proscriptive assertions), as they might be (predictive assertions), and how to achieve desired states (prescriptive assertions).

Proscriptive assertions are the vocabulary unit that aligns most closely with functional specification. Proscriptive assertions describe what the software should do. Each proscriptive assertion has an L binding site (the subject of the assertion) and a conditional propositional logic expression that should evaluate to true. For example:

- There should be at least 5 location update messages transmitted per minute from client to server
 - L binding site: an abstract “client”
 - Condition: “at least 5 location update messages transmitted per minute to server”
- Location update messages should have at most 20 meters of Spherical Error Probable -95 (SEP-95) accuracy
 - L binding site: all location update messages
 - Condition: “less than 20 meters SEP-95 accuracy”
- SymmetricCipher code units must use at least 192-bit keys
 - L binding site: all SymmetricCipher DFUs
 - Condition: “greater than 192-bit keys”
- AsymmetricCipher code units must use at least 8192-bit keys
 - L binding site: all AsymmetricCipher DFUs
 - Condition: “greater than 8192-bit keys”
- AsymmetricCipher code units must use post-quantum secure algorithms
 - L binding site: all AsymmetricCipher DFUs
 - Condition: “has the post-quantum secure property”
- Any data type transmitted between client and server must be encrypted
 - L binding site: any data that flows along an abstract “client-server” edge
 - Condition: “has the encrypted property”

Descriptive assertions describe the current state of things. Whereas proscriptive assertions are typically created before the software runs, descriptive assertions are created during or after software execution. Descriptive assertions have an L binding site (the subject of the observation) and an expression R value (describing the actual observation). For example:

- A location update message was observed on device1 with an SEP-95 of 30 meters
 - L binding site: a location update message on device1
 - Expression: “locAccuracy_sep95 = 30”
- 3 location update messages per minute were sent from client99 to server1
 - L binding site: a dataflow between client99 and server1
 - Expression: “locUpdatesPerMin = 3”
- 1500mbps of network traffic was observed on client99

- L binding site: client99
 - Expression: “traffic_mbps = 1500”
- A SymmetricCipher DFU was initialized with a 128-bit key
 - L binding site: DfuInstance1
 - Expression: “keylength_bits = 128”
- RSA is not a post-quantum secure algorithm
 - L binding site: a DFU that uses the Rivest-Shamir-Adleman (RSA) algorithm
 - Expression: “postQuantumSecure = false”
- Supersingular Elptic Curve Cryptography (SECC) is a post-quantum secure algorithm
 - L binding site: a DFU that uses SECC
 - Expression “postQuantumSecure = true”
- An unencrypted CoT_status message was transmitted from client to server
 - L binding site: a CoT_status message transmitted from client to server
 - Expression: “encrypted = false”

A key point is that these descriptive assertions correspond to the propositional R values of proscriptive assertions. If the L value of a proscriptive and descriptive assertion match, the R valued expression of the descriptive assertion and the R valued constraint propositional logic are checked for consistency. If an observed value violates an assumption assumed to hold in a proscriptive assertion, a constraint violation is generated. The creation of these constraint violations is one mechanism that drives adaptation within the IMMORTALS system.

Predictive assertions describe sensitivities, correlations and relationships between variables. These assertions can be defined before the software executes or they can (more ideally) be inferred as part of the analysis/discovery process. These assertion types have a change-sensitive L value and an impact statement R value.

For example:

- {Increasing image size} {increases bandwidth consumption}
- {Increasing image size} {increases heap utilization}
- {Using a military grade location provider} {increases accuracy}
- {Using a military grade location provider} {decreases location update rate}
- {Invoking the encrypt aspect of a Cipher DFU on some data type} {adds the Encrypted property to the data type}

Prescriptive assertions describe application-agnostic mechanisms to resolve violations of functional spec when they arise. These are useful for defining general-purpose mechanics to repair problems that might develop with software over time. These assertion types have a trigger condition L value and one or more action R values.

For example:

- {If an upper location accuracy bound is exceeded} {use a more accurate location provider}

- {If a lower location update rate limit is undershot} {use a faster but less accurate location provider}
- {If a network bandwidth limit is exceeded} {reduce the frequency of messages transmitted across the network}
- {If a network bandwidth constraint is exceeded} {reduce the fidelity of messages transmitted across the network}

During the process of adaptation, prescriptive assertions can be chained together with predictive assertions to identify viable courses of action. For example, if excessive bandwidth consumption is observed, we could identify a starting strategy to reduce that bandwidth as {reduce the fidelity of messages transmitted across the network}. We could then determine via predictive assertions that lossily reducing the size of an image via resampling decreases the fidelity of the image. We know that the ImageProcessor DFU's *rescale* aspect is one way to do this. Therefore, reducing the bandwidth consumed by the application involves injecting a call to an instance of the ImageCompressor DFU's *rescale* method with a value less than 1.

4.1.1.4 Ontology Toolchain

We have created a suite of tools designed to improve the efficiency of human and software interaction with the IMMORTALS ontology. These tools are described in this section.

4.1.1.4.1 POJO to Triples

Programmatically creating triples is a mechanically easy task due to the existence of a number of mature third party libraries designed for this purpose (most prominently, Apache Jena). However, any instantiation logic must be carefully updated each time the ontology is subsequently refactored. In a frequently changing project, this results in unacceptable delays and brittleness and developers become increasingly hesitant to refactor the vocabulary as it grows due to the complexity involved with even minor changes. IMMORTALS needs a solution that enables quick prototyping and long-term vocabulary flexibility.

This code binding problem is a common one in software engineering, one that has been solved in the past using frameworks like Jackson (JSON), JAXB (XML), XMLBeans (XML)⁵. However, no robust tooling exists for semantic languages like OWL. To that end, Securboron has created a code-to-triples engine that instantiates semantic constructs using Plain Old Java Objects (POJOs). As is the case for many similar frameworks for JSON, XML, and other formats, the pojo-to-triples engine uses annotations to provide additional metadata that addresses the gaps between Java language constructs and semantic constructs:

```
@Triples({
    @Triple(predicateUri=rdfs.subClassOf$,objectId="1"),
```

⁵ JSON is JavaScript Object Notation, JAXB is Jakarta XML Binding, and XML is eXtensible Markup Language

```

@Triple(subjectId="1",predicateUri=rdf.type$,objectUri=owl.Restr
iction$),
    @Triple(
        subjectId="1",
        predicateUri=owl.allValuesFrom$,

objectUri="http://darpa.mil/IMMORTALS/ontology/r2.0.0#MadeUpClas
s"
    ),
    @Triple(
        subjectId="1",
        predicateUri=owl.onProperty$,

objectUri="http://darpa.mil/IMMORTALS/ontology/r2.0.0#hasCodeUni
t"
    )
})
public class AClass extends AnotherClass {

```

For example, the annotated class above would result in the automatic generation of following T-box triples (only a small subset of the actual emitted triples are shown):

```

<http://darpa.mil/IMMORTALS/ontology/r2.0.0/bytecode#AClass>
  a      <http://www.w3.org/2002/07/owl#Class> ;
  <http://www.w3.org/2000/01/rdf-schema#label>
    "AClass" ;
  <http://www.w3.org/2000/01/rdf-schema#subClassOf>

<http://darpa.mil/IMMORTALS/ontology/r2.0.0/bytecode#AnotherClass> ;
  <http://www.w3.org/2000/01/rdf-schema#subClassOf>
    [ a      <http://www.w3.org/2002/07/owl#Restriction> ;
      <http://www.w3.org/2002/07/owl#allValuesFrom>
        <http://darpa.mil/IMMORTALS/ontology/r2.0.0#MadeUpClass> ;
      <http://www.w3.org/2002/07/owl#onProperty>
        <http://darpa.mil/IMMORTALS/ontology/r2.0.0#hasCodeUnit>
    ] ;

```

The idea here is that domain experts familiar with semantic modeling can define these POJOs and programmers can then use them without needing to know what they perceive to be the tiresome details of the semantic web. Critically, concepts relevant to IMMORTALS can be easily instantiated using these POJOs in a consistent way throughout the IMMORTALS project (e.g., two teams performing two different kinds of analyses can easily produce results that align in a useful way).

4.1.1.4.2 Project to Triples

The project-to-triples tool is a Gradle [18] build plugin that scans a project's classpath to automatically extract information about:

- The project's source code
- The project's build/test/execute classpaths
- The project's dependencies on other projects or compiled libraries
- The presence of any DFUs in the application's bytecode

Each of the artifacts on the various classpaths are then analyzed and a high-level model of their bytecode structure is generated from this analysis. The knowledge gained from this initial analysis is fed to the DAS, which uses it to bootstrap the adaptation process.

4.1.1.4.3 Annotation Generator

The annotation generator tool produces Java language annotations for various concepts in the ontology. After generation, these annotations are packaged along with the IMMORTALS POJOs in a single JAR that can be used to then add IMMORTALS metadata to Java projects. These annotations enable programmers to indirectly instantiate ontology concepts via the project-to-triples tool without knowing anything about the format of the triples themselves.

4.1.1.4.4 Annotation Parser

The annotation parser scans compiled bytecode and searches for annotations generated by the IMMORTALS annotation generator. When it finds such annotations, it instantiates the corresponding concept in the vocabulary (e.g., a DFU instance) and binds it to a code scaffold model (e.g., a high-level model of a class). In this way, concepts instantiated via annotation are readily used in later analysis.

4.1.1.4.5 Toolchain Concept of Operation

A hypothetical workflow in which these toolchain components could be used is shown below in Figure 4. Step 1 in this process is the creation of domain knowledge, which is a one-off task performed by subject matter experts familiar with the semantic web. In Figure 4, two simple domains are shown (in reality there are half a dozen different vocabularies used by IMMORTALS).

[illegible]

Step 6 of this workflow involves decompiling and analyzing the compiled software via the IMMORTALS Gradle plugin. Step 6a, bootstrapping, involves a quick static analysis on the software (to build the bytecode scaffold model) followed by an annotation analysis (which instantiates any necessary ontology concepts and links them to the scaffold model). Step 6b is a dynamic analysis step based on DroidScope. Step 6c illustrates a hypothetical integration with other analysis engines that could theoretically provide inputs useful for adaptation.

Approved for Public Release: Distribution Unlimited.

4.1.2 Generic Program Analysis Based KBs

4.1.2.1 Overall Objective

In order to enable automatic adaptation and evolution in response to resource changes, we first need to understand the resource usage and dependency information of the given application. Examples of resource usage and dependency information are shown in Table 1.

Table 1: List of software metadata that can be discovered by program analysis

Hardware Resource Dependency	GPS
	Phone State (Subscriber ID, Device ID, Line Number, etc.)
	Wifi Status
	Network Status
	Media (Audio, Video, Camera)
	Text Message
Hardware Resource Usage	CPU Usage
	Memory Usage
	Network Bandwidth
	Power Consumption
Software Resource Dependency	Third-Party Library
	Android System Service
	Other Components Within the App

As shown above, resource usage and dependency inferred by our tool contains three categories: hardware resource dependency, hardware resource usage and software resource dependency. Hardware resource dependency represents the hardware features that given app will be using during execution. Hardware resource usage shows the resource consumption quantifications of a number of hardware including CPU, memory, power, etc. And software resource dependency is about third-party libraries and other system services that the app utilizes. We designed and implemented a resource usage and dependency discovery framework that can infer fine-grained hardware and software resource usage and dependency information at different level including class level and DFU level given an Android application. This framework consists of two different subsystems, a dynamic analysis tool which is built on top of Droidscope [20] and a static analysis tool which is a modified version of DroidSIFT [21].

4.1.2.2 Key Techniques

The resource usage and dependency discovery framework relies on static and dynamic analysis. The high level architecture of this whole system analysis based software metadata discovery framework is shown in Figure 5.

The framework is designed to perform three types of program analysis to discover the three kinds of software metadata described in Table 1.

Given an input which is an Android app, this framework will trigger Droidscope for dynamic analysis and DroidSIFT for static analysis. The dynamic analysis is responsible for resource usage quantification where we extract hardware resource usage information. On the other hand, the static analysis is leveraged to extract resource data transition and perform resource dependency inference. The results are collected and sent to our post processing component. Eventually triples (conforming to the KR formalism described in Section 4.1.1 are generated and stored in triple store for the whole IMMoRTALS project.

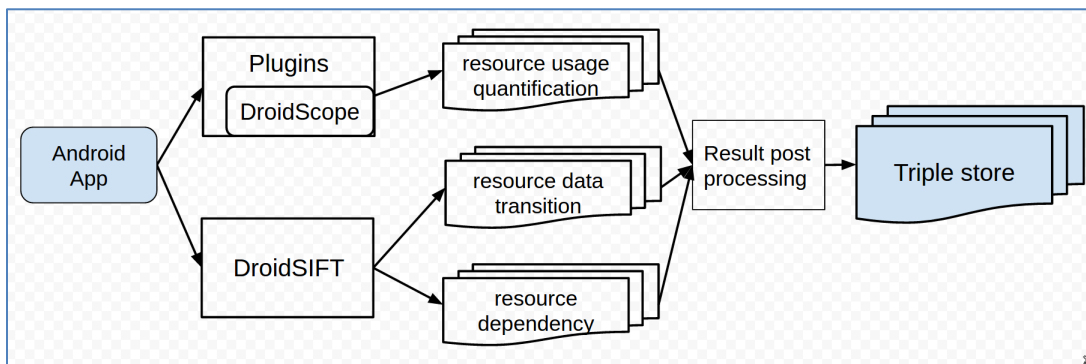


Figure 5: Organization of resource usage and dependency discovery framework

The first resource usage quantification analysis is to understand how much resource is needed in order to execute DFU (e.g., how much memory is required for the DFU). To achieve this goal, we first identify some hooking points during the execution and extract information at runtime. The second resource data transition analysis is to perceive how hardware related data is transmitted across DFUs and non-DFUs.

Droidscope, a VM-based Android app analysis platform, is developed on top of Qucik EMUlator (QEMU). It reconstructs both the OS level and Java-level semantics simultaneously, to further facilitate custom analysis, DroidScope exports three tiered APIs that mirror the three levels of an Android device: hardware, OS and Dalvik Virtual Machine. We modified Droidscope to support the new Android Runtime (ART) and develop a plugin upon it to extract CPU and memory usage information. DroidSIFT is a static analysis tool for Android apps based on Soot, a Java compiler framework. Call graph analysis and data-flow analysis is then performed to extract resource dependency information.

4.1.2.3 Specific Analyses

4.1.2.3.1 Resource Dependency Discovery

The goal of this analysis is to check if the given Android app relies on certain hardware to function properly. The resource dependency includes hardware dependency and software dependency. Currently, we only collect hardware dependency information. In order to collect dependency information, our tool needs to be complete and accurate.

Completeness means we have to cover the whole app to find out dependency information while accuracy requires our tool to have low false positive and false negative. To address these two requirements, we design and modify DroidSIFT, a static Android analysis tool for resource dependency inference.

Table 2: Resource Indicators (pre-defined)

Resources	How to extract
Bluetooth	<code>android.bluetooth.BluetoothAdapter.getDefaultAdapter()</code>
GPS	<code>android.location.LocationManager.requestLocationUpdates()</code>
SMS	<code>android.telephony.SmsManager.getDefault()</code>
Wifi	<code>android.net.wifi.WifiManager.setWifiEnabled()</code>
Audio	<code>android.media.AudioRecord.startRecording()</code>
Camera	<code>android.hardware.Camera.takePicture()</code>
DeviceID	<code>android.telephony.TelephonyManager.getDeviceId()</code>

Table 2 shows the identified indicators of different hardware resources. The indicators are all at Java level with a level of human understandable semantic (i.e., the path to the Bluetooth adapter refers to a Bluetooth resource). The first step is to use method proposed in CHEX⁶ [22] and DroidSIFT to identify all the entry points within the app. During this step, dead code is identified and eliminated. Then our tool statically scans over the whole application starting from those entry points and try to find the listed indicators. Whenever an indicator is found, we conclude that the program is dependent on the indicated hardware resource. For instance, we conclude that the program uses Bluetooth hardware if `android.bluetooth.BluetoothAdapter.getDefaultAdapter()` is found. Finally, since we know where the indicators are located, we can map the locations back to specific functions and classes. We tag those functions as resource consumption functions.

⁶ CHEX is a static analysis tool described in [22].

After the resource dependency inference, we can have the knowledge about what kinds of hardware resources are needed by the program at a fine-grained method level. However, our ultimate goal is to extract resource dependency information for DFUs. The problem now becomes how to map the dependency information back to DFUs. For this purpose, we first extract call graph and class hierarchy information for the whole app. Then we utilize the information to infer if any DFU function eventually calls the resource consumption functions. If yes, we finally label the hardware resource to the DFU.

4.1.2.3.2 Discovering Resource Usage across Processing Thread

The previous section describes how we identify which resources (including hardware) a given software is dependent on. We further discover how that resource is transitively used throughout the program. For instance, given a DFU that is dependent on a specific resource such as a Global Positioning System (GPS) device, how the outputs of that device, for example, the resource or Situational Awareness (SA) data provide by the resource, is used by other parts of the program. We are able to discover this information at two different granularities - call graph level and information flow level.

For this analysis, we assume a-priori defined resource usage source and sink points. These are defined in terms of DFU functions. For example, the following DFU function is a resource usage source since this function extracts current location information from a hardware resource:

```
mil.darpa.immortals.dfus.location.LocationProviderBluetoothGpsSimulated.getCurrentLocation()
```

Similarly, the following function is a resource usage sink, since it writes data onto the network hardware abstraction:

```
java.net.SocketOutputStream.write()
```

Once we have the DFU functions as beginning points and resource endpoints, we traverse through the whole call graph to find all paths between these two points using Breadth First Search (BFS) algorithm. Information flow level data transition, on the other hand, starts from where the sensitive resource related data is retrieved by the program, such as *android.location.LocationManager.getLastKnownLocation()*. And then we perform data-flow analysis for the return value which is the sensitive data and check how the information flows into other parts of the program.

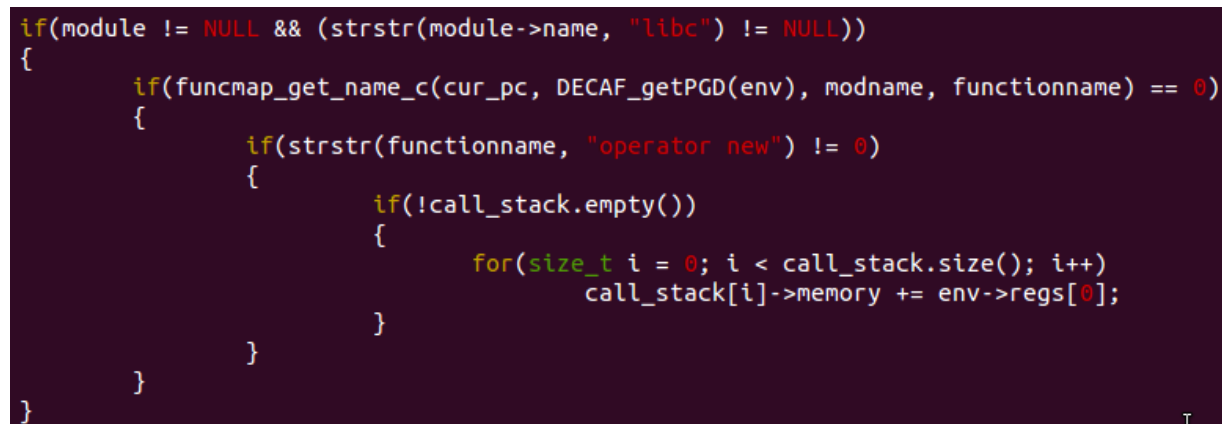
4.1.2.3.3 Resource Usage Quantification

Unlike the above two analyses, resource usage quantification needs to understand precisely how much resource is consumed by the program during execution. Thus, dynamic analysis is needed. To be more specific, we leverage Droidscope as the dynamic analysis platform to extract useful information.

The name of the given Android app is passed to the system for process detection. Only when the right process is launched and detected by the system, we can start the analysis. For CPU usage, we simply register two hooking functions at the beginnings of each instruction and basic block.

```
insn_begin_handle = DECAF_register_callback(
    DECAF_INSN_BEGIN_CB, insn_cb, NULL
);
block_begin_cb_handle = DECAF_register_callback(
    DECAF_BLOCK_BEGIN_CB, block_begin_cb, NULL
);
```

For CPU usage, we first detect at each basic block, if the current basic block is within user space and belongs to an OAT⁷ module. If yes, that means we are within the given android program. We then extract the oat file and analyze the file to retrieve all the method offsets. Once it is done, we can use the current Program Counter (PC) value to determine if we are now at the beginning of a



```
if(module != NULL && (strstr(module->name, "libc") != NULL))
{
    if(funcmap_get_name_c(cur_pc, DECAF_getPGD(env), modname, functionname) == 0)
    {
        if(strstr(functionname, "operator new") != 0)
        {
            if(!call_stack.empty())
            {
                for(size_t i = 0; i < call_stack.size(); i++)
                    call_stack[i]->memory += env->regs[0];
            }
        }
    }
}
```

Figure 6: Memory allocation tracking

method. If true, we store the method information in a call stack that we maintain for getting the complete call sequence and set a global variable named ‘inside_function’ as true. For each instruction beginning, we first check if we are inside a function by using ‘inside_function’ variable. If yes, we then increase the instruction count for each function within the call stack.

For memory usage, we hook the ‘operator new’ operator in libc.so, since this is the function leveraged by JVM for memory allocation.

As shown in Figure 6 if the current context is within a Java level method in the given app, we extract register 0 which is where the memory size requested is stored. We then record the return

⁷ OAT files are created by the Android operating system to speed up the loading of Android Apps.

address and hook the return of the operation to check if the memory allocation is successful. If yes, we add the memory consumption to the current Java level method.

4.1.3 Application ingestion, CM and VM

This section describes how the DAS is initialized, and the roles the CM and VM components play in the overall DAS workflow.

4.1.3.1 DAS Initialization and Application Ingestion

When the IMMORTALS DAS starts, it has already been configured with the generic and application specific KB, CM, VM and AM components. And the triple store has also been populated with relevant application and domain models, as well as application and domain specific knowledge and rules such as *if client side encrypts, the server needs to decrypt* or *one way to reduce network load is to reduce the memory footprint of the payload being sent over the network*.

The initial bootstrap step consists of ingesting the application, and populating the triple store with extracted metadata. The DAS accesses the application – all software, libraries the software depends on, tests, build system etc. - through the file system. The application ingestion and analysis process can be time consuming, by doing the analysis as part of the initialization the DAS attempts to speed up the rest of the workflow.

To initiate the analysis, the filepaths to the application and library build directories are provided to the DAS, which iterates through them until all files are analyzed (see Figure 7). The first step builds the project as is. Building the project generates information about its build structure, which is used by a custom-built Gradle plugin for deriving knowledge about project structure, source organization and dependencies, and available tests. The DAS runs the available tests next, which is used to discover baseline application behavior, resource profiles, and criteria for determining adaptation success (e.g., which tests must pass etc.) Due to the complexities of fielding clients and servers for testing, the server tests were constructed to start up a Java-based client based on code extracted from ATAKLite or actual ATAKLite instances deployed to the Lincoln Laboratory (LL) Evaluation environment. This allowed us to standardize on common unit testing frameworks for independent and cross-application testing by the DAS without having to be aware of the custom evaluation environment or custom testing structure that was created for cross-application testing. As shown in Figure 7, additional KBs, such as program analysis to discover resource dependency were also run as part of the application ingestion.

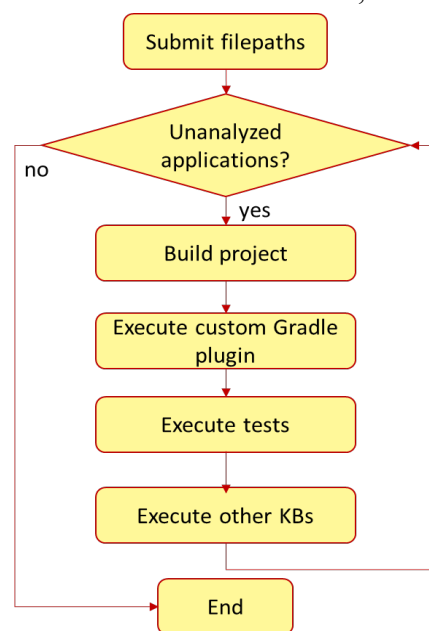


Figure 7: Application ingestion process

Although the DAS could be applied to adapt software in various languages, our platform—the tactical SA application- consists of an Android client and a Java server. The Android community has transitioned to the Gradle build-automation system, and given its openness and modularization, it was easy for us to use Gradle for the Java TAKServer as well. Consequently, we used Gradle’s plug-in framework to build a plugin that could be used to locate source code, build the applications or libraries, test them, and analyze and modify dependencies in an automated manner. As described above, this Gradle plugin plays a coordinating role in the application ingestion process, the results of executing the plugin is made available to other DAS components (KBs, CMs, AMs and VMs) that perform more advanced operations beyond the scope of the build system.

4.1.3.2 Adaptation Steps through Change Manager and Validation

To kickoff adaptation, an RDF file that encodes relevant information including the adaptation pressure (e.g., database perturbations, library upgrades, and new requirements, changes in the deployment model etc.) is submitted to the DAS as shown in Figure 8. At this point, the CM applies the prescribed perturbations to the applications or their execution environments, and the VM executes the validation tests including the tests built into the application’s Gradle project.

The provided perturbations may be such that the intent of the tests are preserved, in which case the DAS decides that no adaptation is needed. If the VM determines that the changes do not preserve the intent, the DAS invokes the AM. After the AM finishes, the DAS invokes the VM to revalidate the application, and reports the results. The RDF submission and the reporting mechanism serve as the interfaces through which independent evaluators interacted with the DAS. For instance, LL’s test harness can submit the RDF describing the change, and receive the results of evaluation of the adapted application as reports.

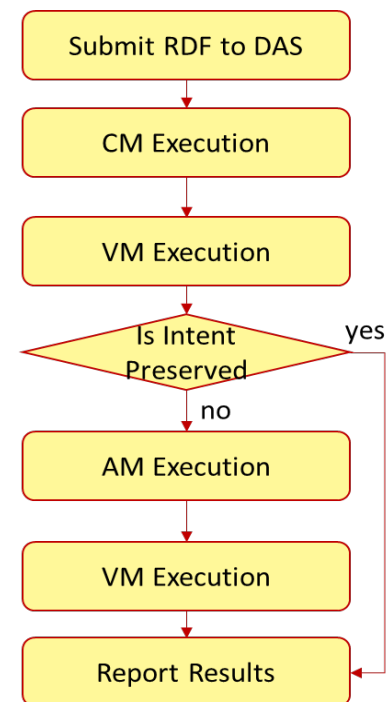


Figure 8: Adaptation

Note that the AM may not find any applicable adaptation, or apply some adaptation that does not satisfy all applicable intents (tests). In the current implementation of the DAS, the workflow does not iterate back to the AM when validation after AM execution indicates intent violation. The AMs are designed to perform best-effort—trying again with the same perturbation from the same initial state is not going to find a different solution.

Note also that a given perturbation may also possibly violate multiple intents, requiring multiple iterative adaptations to address all of them. Iterative addressing of multiple intent violation, which needs to consider conflicts between competing adaptations (i.e., adaptation to fix an intent violation may step on the fix to address a prior violation and invalidate that fix) is not well developed in the current incarnation of the IMMORTALS DAS, remains fairly untested and is an area for future work.

4.2 Handling Changes in Requirements

4.2.1 Component (DFU) Substitution (Phase 1, CP1)

4.2.1.1 Motivation

For years software engineers have used common software libraries and component frameworks to promote modularity and code reuse and thereby reduce software maintenance costs. Though Apache Maven and repositories like Maven Central help manage the dependencies that occur among these modules, the information and query model they provide is limited to dependency coordinates, class names and similar fields. No widespread solution exists to query for components using a rich ontology-backed information model that supports functional properties, resource dependencies and other information. Such a capability is essential in order to support automated software adaptation to changing requirements and resource provisions.

For example, a mobile application may report the device's location using a standard embedded GPS location provider. Some missions, however, may require an alternate provider with safeguards against jamming and spoofing. An adaptive system should be able to query a large annotated software repository for a suitable component substitution when an application's environment changes, as well as modify the application's source code at key locations to initialize and make use of the new component.

4.2.1.2 CP Description

Tactical situational awareness spans a large variety of missions. Such missions might have very different requirements for determining device location. Some deployments may be satisfied with the GPS receiver embedded in most Android mobile devices while others require the specific properties of a Selective Availability Anti-Spoofing Module (SAASM), which in turn requires Universal Serial Bus (USB) connectivity. Some deployments might have to assume a GPS-denied environment where the only option is User Interface (UI)-based dead-reckoning by the device user. Table 3 shows the infrastructure and hardware resource factors that were considered for the ATAK light (ATAKLite) application's location provider capability. A value of true for Ext-USB means that the device has an available USB port *and* an external SAASM device to which the port can attach.

In Phase 1, CP 1 we considered responding to requirements changes surrounding the location information provided by the ATAKLite clients. Changes in this requirement were captured as refinements of the deployment model. Deployment model is the

Table 3: Resource Factors

Resource Type	Abbreviation	Description	Values
Infrastructure	GPS-Sat	GPS satellite availability	True/False
Hardware	GPS-Dev	GPS chipset on Android device.	True/False
Hardware	Ext-BT	Bluetooth chipset on Android device.	True/False
Hardware	Ext-USB	Available USB connection	True/False
Hardware	Has-UI	Client device has a display.	True/False

mechanism IMMORTALS uses to describe the environment (and available resources) where the application is deployed and run. In this CP, the deployment model included a single refinement: Whether or not the deployment required a trusted location provider (Trusted-Required).

A DFU was used to encapsulate the location provider for the device. The team created the five DFUs listed in Table 4 to support this CP.

Table 4: Location Provider DFUs

DFU Abbreviation	Description	Resource Requirements	Resource Provisions
A-GPS	Built-in Android GPS provider.	GPS-Sat + GPS-Dev	Provides basic location information
BT-GPS	Bluetooth based GPS (connects to an external GPS device)	GPS-Sat + Ext-BT	Provides basic location information
USB-GPS	Generic USB-based GPS	GPS-Sat + Ext-USB	Provides basic location information. This DFU is the only option that satisfies the requirement for trusted location services (it should be considered as a proxy for a SAASM provider).
SAASM-GPS	USB-based SAASM GPS	GPS-Sat + Ext-USB	Provides trusted location information
DReck	Manual/dead-reckoning location provider	Has-UI	Provides basic location information

Having actual SAASM hardware as part of the evaluation was infeasible, as well as strictly controlling GPS satellite availability. Hence, the DFUs in Table 4 simulate the necessary functionality. In short, the CP was to select the correct DFU that satisfies mission requirements (specifically to provide basic location services and, optionally, SAASM functionality) for the given resources defined in Table 3. During testing, the system was perturbed by modifying the five resources in Table 3 and the SAASM requirement to generate a perturbation space of 64 options. The following sections describe the key components used to address this CP. Along the way, we also develop the foundation for later CPs.

4.2.1.3 Relevant IMMORTALS Mechanisms and Capabilities

4.2.1.3.1 Overview and Terminology

In Phase 1, CP 1 we focused on resolving a single variation point and the resource evaluation was performed by two modules working together with some overlap. A Java module was developed to locate a suitable DFU component that satisfied all the constraints. Since the deployment model and DFU metadata is represented in RDF in a single triple store, this task was relatively straightforward using SPARQL. Once the component was selected, the Java module modified the source code at the variation point to use the new DFU. Table 5 lists the IMMORTALS concepts and mechanisms that played a role in the initial solution to this CP.

In parallel, a separate Haskell-based program was developed to verify the choice the adaptation module made. The Haskell program was later enhanced to select (versus verify) multiple components and their configuration, and the Java module adaptation module was modified to focus on making code changes and building and testing the application. This phased approach was used to allow additional time to develop the Haskell-based solution, which ultimately leveraged a custom type system, variational analysis and a Satisfiability Module Theories (SMT) solver to evaluate more complex groups of variation points, a task that would not be straightforward or feasible using SPARQL. The following sub-sections describe the challenge problem and how the above components were used to solve it, followed by the CP 1 test design and results.

Table 5: IMMoRTALS concepts and mechanisms relevant to Phase 1 CP1

Feature Model/Variation Points	Provides a traceable path from each application feature to one or more variation points within the source code. Each variation point represents a set of labelled locations (i.e., a template) where a component that implements the application feature can be initialized and put to use.
Annotated Discrete Functional Units (DFU)	An annotated repository of code where each <i>Discrete Functional Unit</i> (DFU) is associated with metadata that describes its functionality and resource requirements. For the Java language, each DFU corresponds to a Java class, but the concept may map to different constructs in other languages as appropriate.
Deployment Model	Describes the available hardware and infrastructure and requirements for a specific application deployment. The latter component is sometimes referred to as a mission model.
Adaptation Module	A software module that resolves the variation points to concrete DFUs that collectively satisfy the deployment model. This module consists of two components that work together: (1) A component that is responsible for choosing and configuring the appropriate DFUs for the current deployment; (2) A component that is responsible for modifying the labelled locations at the appropriate variation points so that each new DFU is correctly initialized and put to use in the baseline application.

4.2.1.3.2 Feature Model and Variation Points

Applications may be analyzed according to the features they provide. For example, an Android-based situation awareness application may provide several key features, including the device's latitude/longitude pair, elevation and camera imagery. Features may be refined according to various qualities of service, such as image resolution or location reporting rate. Moreover, not every feature may be required in some missions. Software engineers assemble one or more discrete components (or DFUs) to implement these features. An important requirement for building long-lived software is recognizing that this process should not be limited to decisions that occur only during the application initial design. Requirements and available resources continue to change throughout the life of the software; hence, to lower the cost of software maintenance, the IMMoRTALS DAS provides an efficient means to represent these key choices within the

application, including where they occur in the source code, to which features they are linked and other information shown in Figure 9.

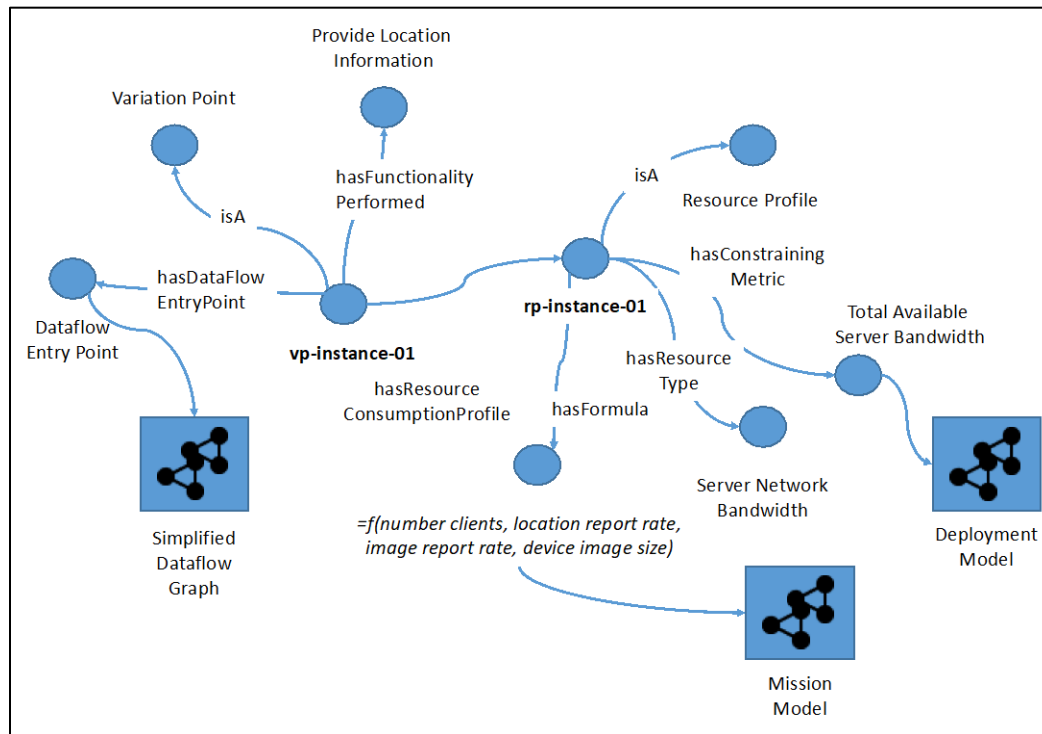


Figure 9: Variation Point Example

All information is represented as RDF triples using the Apache Jena library. The names of subjects and predicates are simplified in Figure 9 (some of the information is used in CP 2 discussed later):

- **vp-instance-01** (in **bold**) is an instance of a variation point. It represents a call site where a DFU that provides location services may be initialized and put to use.
- The variation point is connected to a processing pipeline of DFUs that combines location information and imagery to form a CoT XML message that is then sent to the TakServer. A model of this architecture is created in RDF to support more advanced adaptation techniques in Phase 1, CP 2.
- In addition to a resource profile associated with each DFU, a resource profile may also be associated with each variation point (as shown in Figure 9). Each resource profile is associated with exactly one resource type (e.g., Server Network Bandwidth in Figure 9). The availability of this resource is constrained by the deployment model and its consumption level can be evaluated using the linked formula. The parameters for the

formula are resolved using values in the deployment model. Resource profiles are used in CP 2, Phase 1 to support application adaptation in bandwidth constrained environments.

Because the information is represented as an RDF graph, it is straightforward to create a SPARQL query that finds a suitable DFU. Once a DFU is found, however, the code at the variation point must be modified to make use of it. A simple code templating engine (specifically Velocity) was used to mark the call site at key places where a location provider could be initialized, invoked and released. Each DFU, in turn, was marked with similar labels to indicate which methods correspond to each label at the variation point.

For example, the ATAKLite location providers follow a common *initialize-use-cleanup* usage pattern. Location providers are used at a call site that conforms to Android's IntentService lifecycle, which is an event-driven model that provides service implementations with well-defined integration points. The location provider's usage pattern must be mapped to this event-driven pattern at the call site in order to use the component correctly. Figure 10 depicts this mapping.

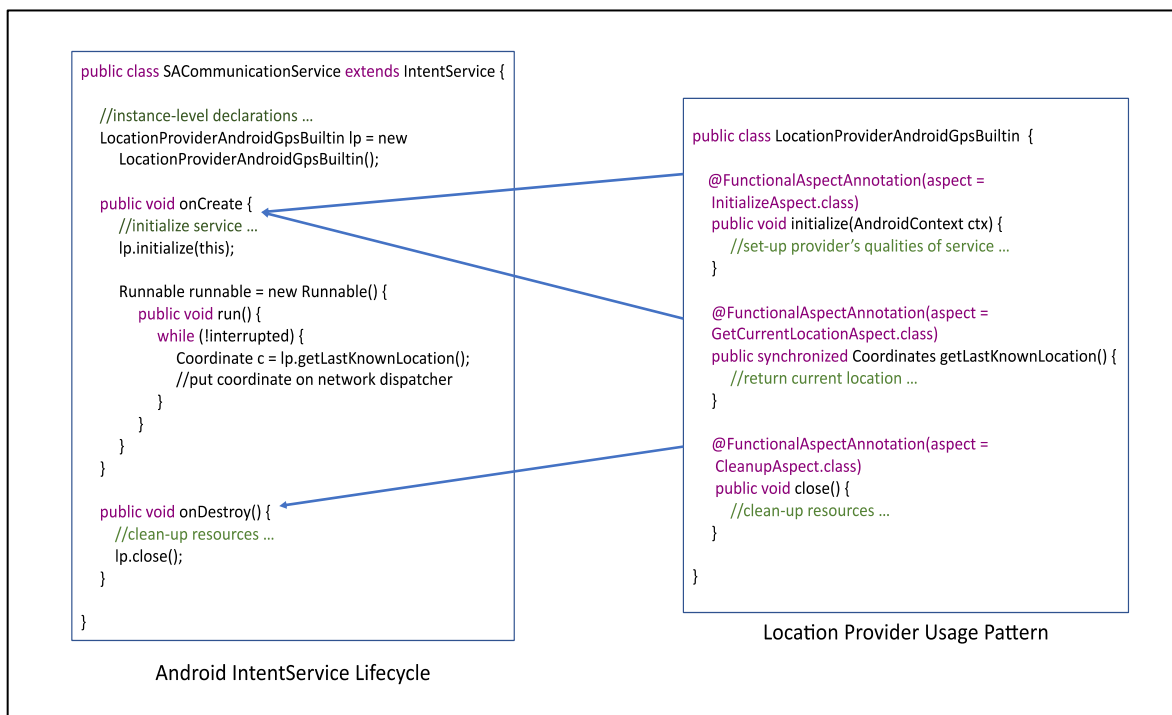


Figure 10: Usage Pattern-Call Site Mapping

A suitable target for the location provider's *initialize* step is the *onCreate* event. We see a similar mapping that must take place for the other location provider usage pattern steps. These must be mapped to appropriate regions of code within the IntentService lifecycle so that they are invoked the correct number of times and at the right time.

Object-oriented programming and the use of interfaces enable code substitution through polymorphism, but this approach only works for constrained areas of functionality where industry-wide standardization efforts have been adopted. Short of this standardization, developers of libraries that provide similar services will conceptualize the domain problem differently and thereby create APIs with usage patterns that are very different from one another.

This simple scheme was adequate for the needs and scope of CP 1, but it does not provide a robust general solution for addressing this key issue in building long-lived software.

4.2.1.3.3 Annotated Discrete Functional Units (DFUs)

Figure 11 depicts the metadata captured for each DFU. The triples describe the DFU's resource dependencies, overall functionality, and arbitrary single-valued properties, which can be used to represent extra-functional refinements. DFU metadata is collected from the software engineer through Java annotations as shown in Figure 12.

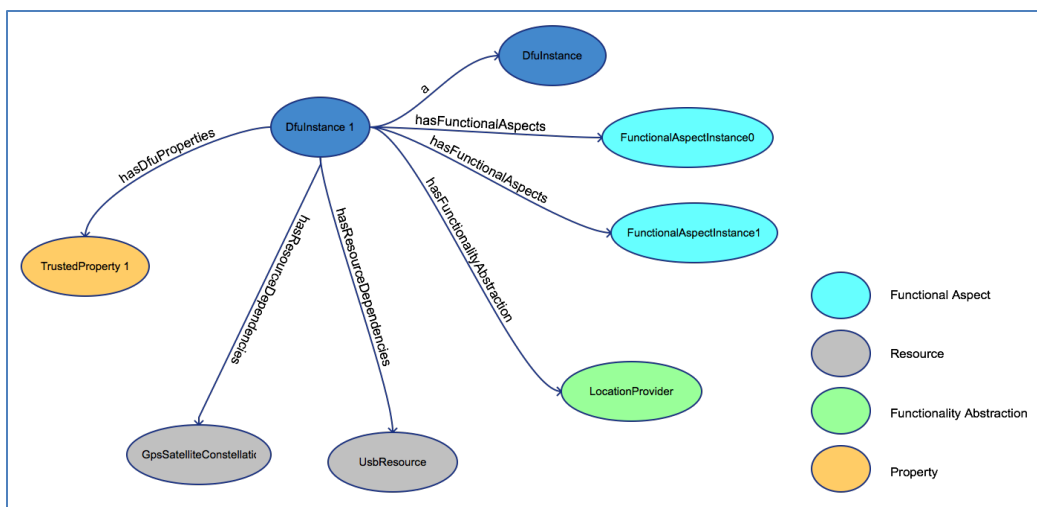


Figure 11: DFU Metadata

```

25 @DfuAnnotation(
26     functionalityBeingPerformed = LocationProvider.class,
27     resourceDependencies = {
28         GpsSatelliteConstellation.class,
29         BluetoothResource.class
30     }
31 )
32 public class LocationProviderBluetoothGpsSimulated {

```

Figure 12: DFU Annotations

During build time, the DAS processes the DfuAnnotation annotations and converts them to triples in the form of .ttl files. At the end of the process, the .ttl files, together with the deployment model and the results of static and dynamic code analysis (developed in other CPs described in this document), are ingested into a running Fuseki Jena triple store to provide a complete view of the software environment. The value of a unified metadata repository that supports ongoing adaptation is a key contribution of IMMORTALS.

4.2.1.3.4 Deployment Model

Metadata contained in the deployment model is a key aspect of adaptation. This information includes:

- Hardware resources and their capabilities (e.g., camera properties, USB connectivity, etc.).
- Infrastructure details and constraints, including available network bandwidth and the number of clients a mission must support.

The Deployment Model graph begins with the triples depicted in Figure 13. The deployment model specifies available resources, like server and Android platform instances. An instance in this context refers to an instance of the platform type versus a specific device. For example, an Android platform type may specify a Marshmallow device with the following hardware: a USB port, a Bluetooth transceiver, a physical UI, and an embedded GPS receiver. This information is used to identify any gaps between resources dependencies declared in DFUs and the capabilities of available hardware on a given mission. <Functionality Spec 1> in Figure 13 indicates that

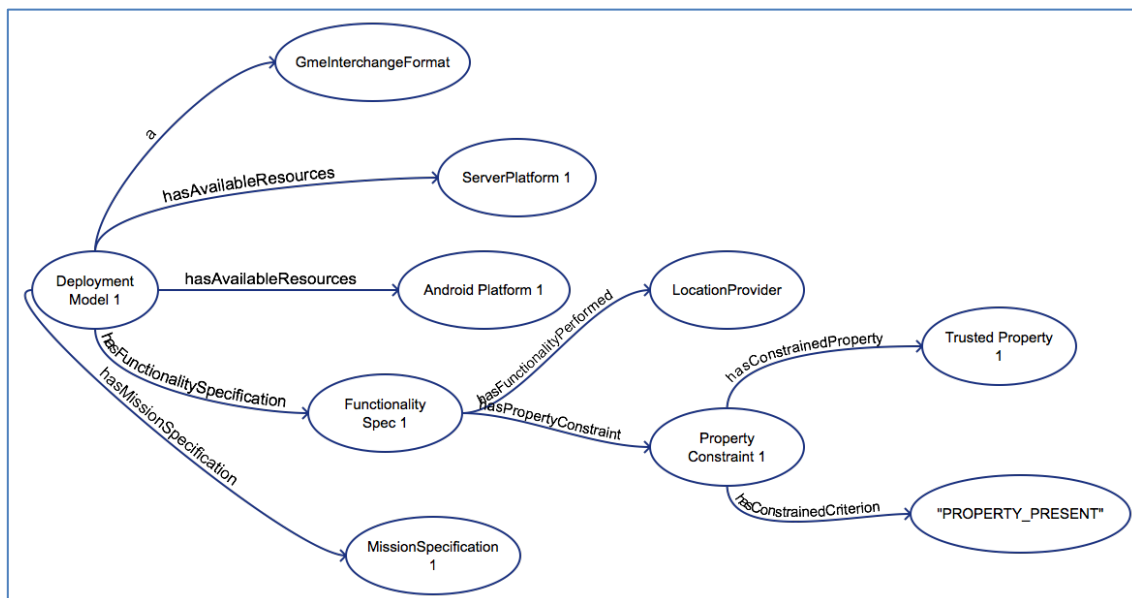


Figure 13: Deployment Model

<LocationProvider> functionality is required and has the <TrustedProperty 1> constraining property associated with it.

Mission specification instances carry many of the mission's constraints and requirements, including:

- Minimum Position Location Information (PLI) report rate
- Number of device clients the mission must support
- Minimum image report rate
- Available total bandwidth at the server

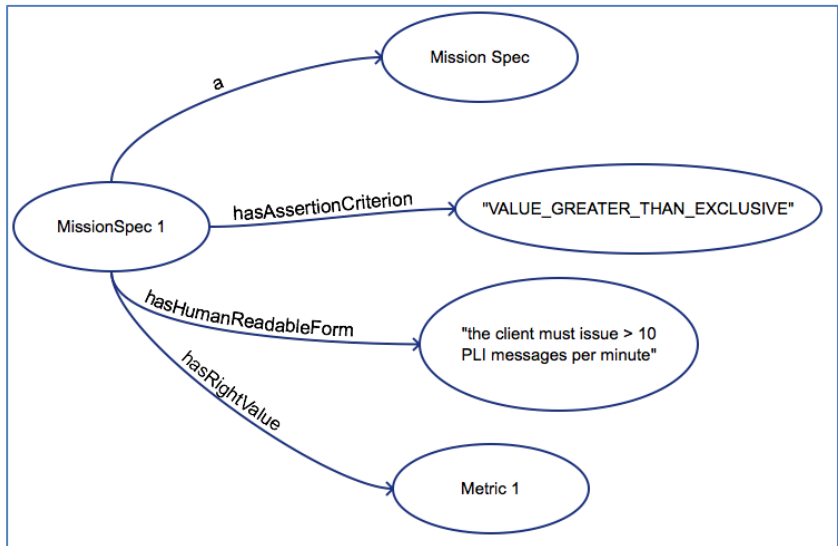


Figure 14: Mission Specification

These are used in Phase 1, CP 2. Each mission specification has two overall parts: (1) A metric instance, which includes a value and the units for that value; and (2) The relationship the mission has to that metric (the assertion criterion). This is shown in Figure 14. <Mission Spec 1> has an assertion criterion with respect to <Metric 1>. <Metric 1> triples are depicted in Figure 15. Together, these triples encode the mission requirement that ATAKLite clients must submit over 10 location reports per minute.

Most of the predicates in these triples are self-explanatory. The *hasLinkId* predicate in Figure 15 is used in Phase 1, CP 2 to

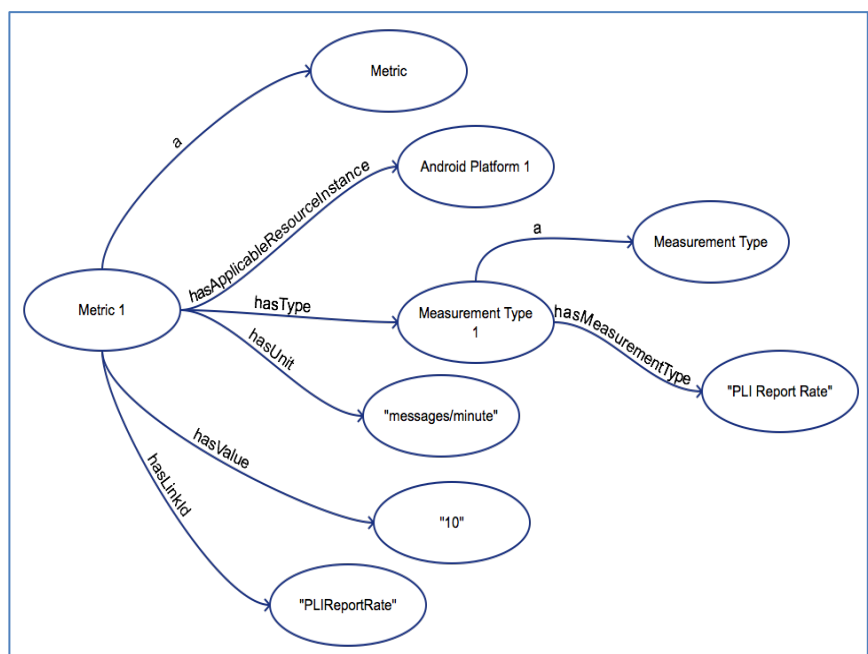


Figure 15: Metric Definition

allow the metric's value to be substituted in mathematical expressions at the variation point that compute resource usage. This mechanism makes it possible to verify resource constraints when mathematical models are available and expressed using the same terms from the mission specifications.

4.2.1.3.5 Adaptation Module

```

1  "SELECT DISTINCT ?dfu ?className ?groupId ?artifactId ?version " +
2    "?functionalityUri " +
3  "WHERE { " +
4    "    GRAPH <" + bootstrapUri + "> { " +
5    "        ?dfu a dfu:DfuInstance . " +
6    "        ?dfu im:hasFunctionalityAbstraction ?functionalityUri . " +
7    "        ?dfu im:hasClassPointer ?classPointer . " +
8    "        ?class a bytecode:AClass . " +
9    "        ?class im:hasBytecodePointer ?classPointer . " +
10   "        ?class im:hasClassName ?className . " +
11   "        ?jar a bytecode:JarArtifact . " +
12   "        ?jar im:hasJarContents ?content . " +
13   "        ?content a bytecode:ClassArtifact . " +
14   "        ?content im:hasClassModel ?class . " +
15   "        ?jar im:hasCoordinate ?coordinate . " +
16   "        ?coordinate im:hasVersion ?version . " +
17   "        ?coordinate im:hasArtifactId ?artifactId . " +
18   "        ?coordinate im:hasGroupId ?groupId . " +
19   "        filter (?functionalityUri = " + functionalityUri + ") ";
20
21  for (String property : propertyUris) {
22      query = query.concat(
23          "?dfu im:hasDfuProperties ?propertyInstance . " +
24          "?propertyInstance a " + property + ". ";
25      )
26
27      query = query.concat(
28          "MINUS { " +
29          "SELECT DISTINCT ?dfu " +
30          "WHERE { " +
31          "    GRAPH <" + bootstrapUri + "> { " +
32          "        ?dfu a dfu:DfuInstance . " +
33          "        ?dfu im:hasFunctionalityAbstraction ?functionalityUri . " +
34          "        ?dfu im:hasResourceDependencies ?resource . " +
35          "        FILTER(?resource NOT IN ( " + resources +
36          "        ) " +
37          "    ) " +
38          "    } " +
39          "    } " +
40          "    }";

```

Figure 16: DFU Search Query

In Phase 1, CP 1, the DAS system iterated the functionality labels described in the deployment model and, for each functionality, executed the SPARQL query shown in Figure 16, which filters DFUs based on the required functionality, properties and resources. Each candidate DFU is

verified using resource compatibility checker (described in Section 4.2.1.3.6) and a final candidate is chosen (if more than one candidate is available, an arbitrary choice is made). The DAS next iterates each variation point for the functionality and modifies the source code at the linked file location using the velocity template engine. The final step is to compile and build the application and invoke the tests defined in the validation modules. These tests are described in the evaluation section below.

4.2.1.3.6 Resource DSL

Resource DSL refers to a *domain-specific language* for describing the resource requirements and provisions of components within a system, and of the environments those components are designed to be used in. Programs in this DSL are executable specifications called *resource models*. Together, the resource descriptions expressed in the DSL and the supporting DSL execution tool, provided the IMMORTALS capability to check resource compatibility of adapted software as well as to aid resource compatible adaptations.

The high-level goal of the DSL is to provide a language for (1) specifying the resource models of software components within the system, (2) composing the models of components into larger resource models that describe the resource usage of an entire system, and (3) describing the resource constraints of environments where the system is expected to run and/or resource requirements the system is expected to satisfy.

Conceptually, the *resource model* of a software component captures the resources that a component *requires* and potentially *consumes*, and the resources the component *provides*. For example, the resource model of a USB-based location-provider DFU might require and consume a USB port, require (but not consume) GPS capabilities, and provide a location service that other components may depend on. The DFU cannot be used within the system if its requirements are not satisfied. The resource requirements of one component may be satisfied by the resource provisions of another component, or by the resource provisions of the underlying resource environment.

A resource model is executed in the context of an initial *resource environment*. The resource environment is conceptually a hierarchically organized global state. The environment may include the hardware capabilities of the platform that the system will run on, consumable resources such as network bandwidth or available ports, as well as any libraries, services, or other capabilities. Resource models will reference, check, modify, consume, and add resources to the environment during execution.

After execution, the resource environment is checked against a given *mission requirement*, which is essentially just a final model that must be satisfied after execution of the model. The mission requirements include the capabilities that the subject system must have in order to successfully fulfill its duties. For example, the mission requirements for Phase 1, CP 1 would include a location provider service and potentially more specific checks of the particular kind of location provider service offered. Thus only systems (DFUs) that provides this service would be compatible with the mission requirements.

All resource models (including the mission requirements, which is just a special distinguished resource model) and the resource environment are potentially *variational* [23, 24, 25]. This means that they can be configured to require, consume, and provide different resources based on a set of *configuration options*. The configuration options in a resource model may correspond to many different kinds of variability in the system, such as optional features in the feature model, differences corresponding to different deployment environments or platforms, or variable quantities associated with particular resources, such as the amount of network bandwidth available (which might in turn vary depending on other configuration options, such as where the system is deployed).

Since systems consist of many components, each represented by configurable resource models, the configuration space of a system grows very large very quickly (exponentially in the case of independent, Boolean configuration options). In Phase 1, we execute only fully configured models to validate whether they work in the given (fully configured) resource environment and satisfy the mission requirements. However, in Phases 2 and 3, we extended the semantics of the Resource DSL to explicitly consider the variability of resource models into account using *variational execution* [24, 25, 26, 41, 40], which tracks variation locally and shares commonalities wherever possible, to support efficiently searching the huge space of potential configurations.

4.2.1.4 Evaluation

The DAS CP 1 validator implements the following set up:

1. After the ATAKLite client application is adapted and compiled, two instances are launched, along with the TakServer server, which shares location reports across all clients.
2. Since the clients were run in Android emulators, they did not have access to GPS devices; hence, the DFU location providers were built to provide simulated values. The name of the DFU that is selected during adaptation is stored in the CoT message in the CoT Type field.
3. In the test environment, each ATAKLite client logs its sent and received messages using the Log4J network logger, which stores its log entries in a central location shared by all ATAKLite instances.
4. A DAS validator was built to inspect the mission requirements and available resources in the deployment environment to determine which tests need to be run. The tests in turn inspect the logs, including the CoT Type field which indicates which location provider is in use by each client, for evidence that the correct adaptation was performed.

The tests shown in Table 6 were applied by the validator (the core tests are required to pass for basic situational awareness functionality, while the others verify a specific combination of conditions that depend on the way the system was perturbed).

4.2.1.4.1 Internal Evaluation

For the internal evaluation we tested a subset of the possible combinations that could be chosen by the evaluator and validated the test results. All tests were utilized to ensure no Data Acquisition Units (DAUs) were giving off unexpected successes. Manual examination was also performed for one instance of each DFU usage to ensure that the adaptation performed as expected and that the DAU behavior was correctly coming from the selected DAU. The executed tests behaved as expected, with no deviation from the anticipated DAS, Application, or Test behavior.

4.2.1.4.2 External Evaluation by the Voice of the Offence Team

The challenge problem was evaluated by Voice of the Offense team from the MIT Lincoln Laboratory (MIT LL) using the test procedure described in this section. The platform under test includes the ATAKLite and TakServer applications, together with the connecting network. In this CP, only the ATAKLite application was adapted.

Two sets of tests were performed. The first set was meant to verify the basic functionality of the test apparatus itself and will not be covered here. The second set of tests perturbed the factors shown in Table 6 within the space of 64 possible perturbation profiles.

Table 6: Evaluation Tests

	Test	Description	Core
1	client-location-produce	Test where each client produced a situational awareness (SA) report.	Yes
2	client-location-share	Test whether each client received an SA report.	Yes
3	client-image-produce	Test whether each client produced an SA image.	Yes
4	client-image-share	Test whether each client received an SA image.	Yes
5	client-location-source-trusted	Test whether client used a trusted location provider.	No
6	trusted	Test whether all clients used a trusted location provider during the entire test.	No
7	source-usb	Test whether client used a USB connected provider.	No
8	source-bluetooth	Test whether client used a Bluetooth connected provider.	No
9	source-androidgps	Test whether client used an embedded GPS receiver.	No
10	source-manual	Test whether used a dead-reckoning provider.	No
11	bandwidth-maximum-validator	Test whether the bandwidth consumed at the server (sampled every second) is below the required mission threshold. (applies to CP 2 only)	No

Table 7: Test Stages

Stage	Description
Baseline A	This stage represents the un-adaptive baseline (i.e., the default baseline profile is used).
Baseline B	This stage presents new mission requirements or changes to the available resources to an un-adaptive platform; i.e., the platform is expected to fail.
Challenge Stage	This stage presents the same mission requirements as Baseline B but allows the IMMORTALS DAS to adapt the ATAKLite client.

The following discussion makes reference to a default baseline perturbation profile which is defined in Table 8. Each perturbation was executed in three stages as shown in Table 7.

Table 8: Default Perturbation Profile

Factor	Value
GPS-Sat	True
GPS-Dev	True
Ext-BT	False
Ext-USB	False
Has-UI	False
Trusted-Required	False

Since the un-adaptive client-server applications and parameters for Baseline A never change it was not necessary to run for every test. The focus for evaluation was to determine how Baseline B and the Challenge stage responded as the input parameter space was changed. For each test, the IMMORTALS system returned either:

- **Solution Possible**

A variant of the platform can be created to function correctly in the deployment environment; note that this variant may or may not require adaptation.

- **Solution**

Infeasible No adaptation is possible given the current resource profile, requirements and code repository.

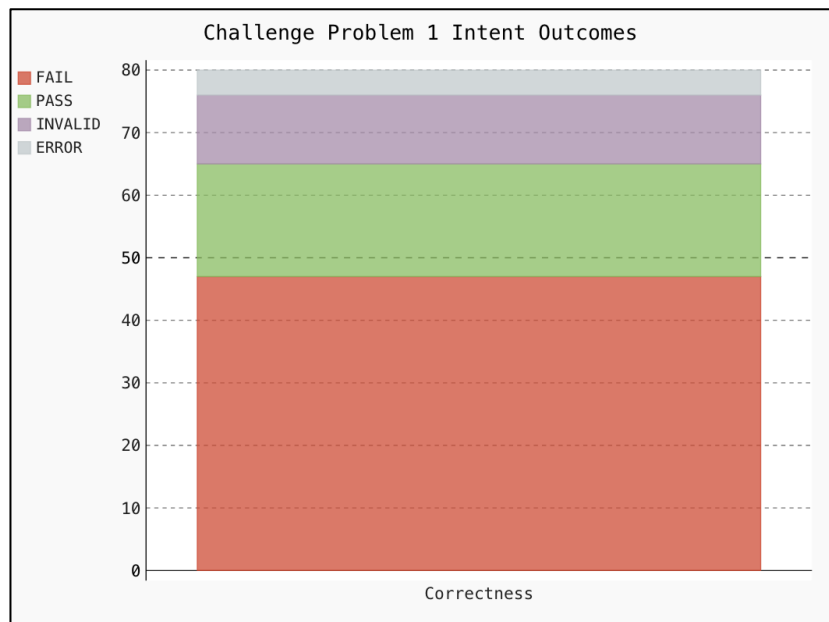


Figure 17: CP 1 Intent Outcome

A total of 80 tests were evaluated with the majority, 65 tests, completing successfully (see Figure 17). In this context, successful completion is defined as resulting in a 'Pass' or 'Fail' without an unexpected error occurring in the DAS or the test environment. Out of the remaining tests a total of 4 experienced test errors which caused them to terminate early, the remaining 11 tests are invalid due to the fact that the perturbation was not sufficient to cause test stage Baseline B to fail. As shown in Figure 17, of the 65 completed tests 47 were shown as failed in terms of intent outcome. An outcome of 'Fail' in this case indicates there is a mismatch between the actual output and the desired solution (i.e., a DFU that satisfies all perturbed requirements was found). However, given the DFU inventory, not all random variations in location provider requirements may not be satisfied. The MIT LL report summarized these findings as follows: "All 47 of the tests identified as FAIL represent cases where IMMORTALS was unable to construct a new solution to satisfy the input mission constraints. The large number of tests where a solution was not found is not surprising in this case.

A successful scenario typically requires more than one constraint (interface and location providing service) to be available. Since the test generation process selects each parameter independent of one another there is an increased likelihood that it will choose constraints which cannot be satisfied." In other words, under the experiment's inherent design, 30 of the 64 possible perturbation profiles (roughly 47%) lead to infeasible outcomes. Charts produced by MIT Lincoln Laboratories reflect these simply as failures. This highlights that it is not meaningful to randomly perturb the input space as MIT LL did.

4.2.2 Handling New Requirement on Client-Server Dataflow

4.2.2.1 Motivation

SA data flowing between Android clients via the server is a key aspect of the tactical SA platform. As a result, the client-server application is susceptible to evolutionary pressures that changes how data in motion is handled. Examples include adding encryption or compression, or changing cryptographic or compression techniques and parameters based on updated best practices. As a historical justification consider the evolution of cryptographic standards and best practices over time. Data Encryption Standard (DES) became a Federal Information Processing (FIPS) standard in 1975 but was easily brute forced by the late 1990s; its successor Triple DES (DES3) became a standard in 1999 but was subsequently superseded by Advanced Encryption Standard (AES) in 2001. Similar trends have been observed for asymmetric algorithms, with the recommended RSA key length increasing from 512 bits at inception to 2048 bits as of 2016 (with a target key durability of 2030). One key aspect that motivated this CP is that the impact of such changes cross the process boundaries – adding or changing the data on the client will require appropriate changes on the server side that needs to decrypt or uncompress the data sent by the client.

4.2.2.2 CP Description

This CP is based on the ATAKLite clients communicating with the MARTI⁸ server. The CP assumes the following inputs:

- A description of the client-server architecture expressed in terms of the IMMORTALS ontology and a corresponding constraint that mandates a property of the data in motion—in this CP we used a specific constraint mandating confidentiality of the client-server data transmission.
- Description of locations in code using the IMMORTALS ontology that transmit and receive information across the process boundary and must be provided manually because it is difficult or impossible to infer linkages between processes using bytecode analysis.
- Domain knowledge about cryptographic best practices. This is a pruned-down representation of the information in FIPS 140-2 expressed in terms of vocabulary from the IMMORTALS ontology or other cryptographic cipher and key combinations that exercise the adaptation process adequately.
- Properly annotated cryptographic DFUs. We seeded a mock code repository with several cryptographic DFUs annotated with semantic code bindings that link code units to the IMMORTALS ontology. The repair process involves traversing this repository to identify code units useful for application repair.
- Usage patterns for cryptographic code units. In addition to the DFUs themselves, we also require a description of how to use them in practice (again, expressed using the

⁸ MARTI is the unrestricted and lighter version of the Team Awareness Kit Server (TAK Server).

IMMORTALS ontology). For example, a streaming cipher would have a different usage pattern than a blob-based one.

4.2.2.3 Relevant IMMORTALS Mechanisms and Capabilities

4.2.2.3.1 Knowledge Builders

We have used a combination of static, dynamic, and various semantic analyses to extract the knowledge needed for adapting the client and server side code for this CP. Of specific interest is identification of the path SA data takes through code when it is sent from the client and is received at the server, and the functions involved. This information is needed to inject new symmetric/complementary functionality such as encryption and decryption at the appropriate points in the data paths across process boundaries. The analysis mechanisms operate on Java bytecode.

4.2.2.3.1.1 Program Structure and Control Flow Analysis

We have used several flavors of static analysis to derive knowledge about the structure of the program (scaffold model) and its control flow. To construct the scaffold model, IMMORTALS traverses a static analysis classpath (typically equivalent to the runtime classpath of an application being analyzed) and uses the Soot [42] and ASM⁹ [43] libraries to inspect the bytecode it finds there. From this initial and cursory inspection, IMMORTALS populates an intermediate triple-based representation of any identified classes, methods, fields, and annotations. For methods, the argument and local variable definitions are also stored. These triples are placed in a graph associated with the analysis problem instance.

After constructing the structural scaffold, IMMORTALS constructs a Control Flow Graph (CFG) for each method on the analysis classpath. A CFG is made of *basic blocks*, which are sequences of instructions for which the following are true:

- Only the first instruction the basic block may be the target of a conditional JUMP or GOTO
- Only the last instruction in the block may be a conditional JUMP, GOTO, RETURN, or THROW
- If the first instruction in a basic block is executed, all instructions in the block must be unconditionally executed (barring the occurrence of an exception)

Directed edges in the CFG indicate possible control flow paths. For JVM bytecode, most nodes will have zero (if the node ends in a return or throw), one (if the node ends with a GOTO instruction), or two (if the node ends with a conditional JUMP instruction) outgoing edges. However, some nodes, such as those ending in a LOOKUPSWITCH instruction may branch to an arbitrary number of locations. The number of inbound edges is not restricted.

Once CFGs for all methods on the analysis classpath are constructed, the KB dumps a representation of each CFG into an intermediate representation backed by an ontology. These

⁹ ASM is a general purpose Java bytecode manipulation and analysis framework—it is not an acronym.

CFGs can then be linked to perform higher-order analysis (e.g., determining the existence of a viable control flow path that transcends method invocation).

Static control flow analyses such as the ones described above provide a solid foundation for understanding high-level program behavior. For example, we might use the analysis above to determine how the various code units in a program interact at a high level (e.g., the function `processImage` calls the static function `transmit`, which in turn calls `write` on an `OutputStream` retrieved from a `Socket` instance). We might also be able to determine whether a viable execution path exists between two code units. However, for JVM languages (and in general), static analysis has several practical shortcomings prevent it from being directly useful for application repair:

- Noise. Many more possible control flow paths may appear to exist in a program than are ever executed in practice. This is especially true if the analysis considers exceptional execution paths. These spurious edges add significant overhead to any higher-order analysis that is based upon tracing viable control flow paths (e.g., dataflow analysis).
- Ambiguity. For all invocation types except `INVOKESTATIC` (`INVOKEINTERFACE`, `INVOKEVIRTUAL`, `INVOKEDYNAMIC`, and `INVOKESPECIAL`—with caveats), the actual callsite of an invoke instruction is not known until the application is compiled. For example, an `INVOKEINTERFACE` will operate on whatever implementation of the interface is on the stack at runtime. For applications in which a single implementation of the interface is known to exist on the classpath this can be reduced, but for other cases all possible implementations must be considered. Similar issues arise with `INVOKEVIRTUAL` (polymorphism allows subclasses to override parent functionalities).
- Type II errors. More problematic than the noise and ambiguity issues presented above (which together contribute to Type I or false positive issues) is the tendency of static analyses to mistakenly omit CFG edges entirely. This happens whenever a reflective method invocation occurs because the reflective callsite cannot be determined at the time of compilation (for example, a reflective call might invoke a method whose name is stored in a text file only provided at runtime). Although this seems like a very abstract corner case, it is seen often within frameworks such as Spring.

Static CFG analysis is a step toward the goal of identifying how data, specifically our case SA data moves through the code. Because accurate understanding of how SA data flows through code is paramount for correctly handling desired properties such as confidentiality, we augment the initial static analysis step with dynamic analysis. This dynamic analysis step was manually performed for this Phase 2 CP, but in Phase 3 we have successfully integrated this into an integrated KB that identifies dataflows of interest such as the paths taken by SA data that transcend process boundary.

4.2.2.3.1.2 Data Flow Analysis

Dataflow analysis is frequently intractable in practice due to the combinatorically explosive search space of all possible control flows within nontrivial programs. One way to limit the search space is to exclude 3rd party software. But this is severely limiting considering there very well might be IMMORTALS annotated software (DFUs) in the application's dependencies. Another approach is

to use a dynamic analysis approach by instrumenting the user's application to generate call traces from which IMMoRTALS can derive data flows. This also does not guarantee that all flows touching IMMoRTALS managed component will be discovered. To address these concerns, we introduced the concept of *Reach Factor*—a tunable parameter that configures a desired balance between coverage and efficiency by specifying how many method invocations from the dynamic analysis-generated call trace the dataflow analysis will *reach* into. This hybrid dataflow analysis is illustrated in Figure 18 with a Reach Factor of two. As shown the nodes in the call trace have the reach counter 0. Nodes one hop from a node with reach counter value x has the reach counter value $x+1$. The analysis stops at nodes with reach counter value equal to the Reach factor. Note however, that the reach counter is reset (instead of incremented) when the next hop node represents IMMoRTALS managed code (e.g., annotated DFU), resulting in expanding the scope of analysis to that node's neighborhood. The end result is a dataflow analysis approach that will not cause an out-of-memory error and can scale with the user to increase its coverage into the application's dependencies.

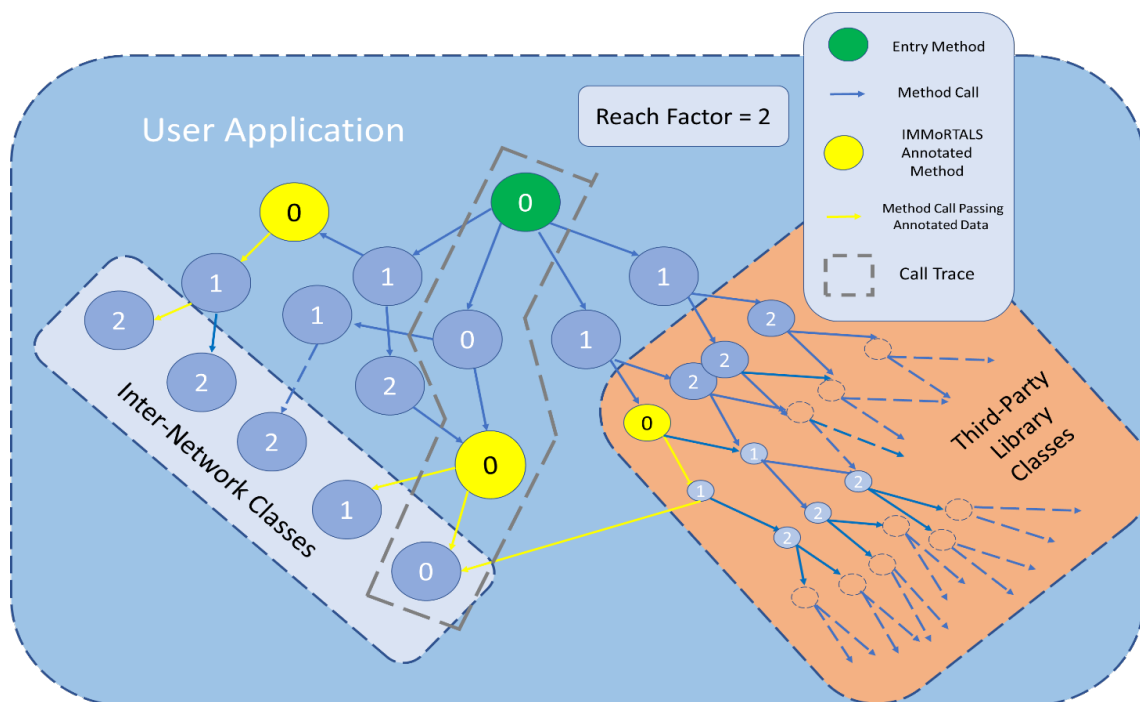


Figure 18: Illustration of Reach Factor driven hybrid analysis

By construction, the hybrid analysis and the various analyses it builds upon generates a lot of metadata about code in addition to the dataflow information. Combining them with the metadata extracted by generic KBs (such as the build and package structure) presents an opportunity to

By construction, the hybrid analysis and the various analyses it builds upon generates a lot of metadata about code in addition to the dataflow information. Combining them with the metadata extracted by generic KBs (such as the build and package structure) presents an opportunity to enrich the dataflow information with information about various types of boundaries that can be helpful in locating call sites and code blocks relevant for adaptation, including:

1. The boundaries between the methods within the user classes and applications
2. The boundary between the user application and what classes belong to its third-party dependencies
3. The boundary between the user application and what classes belong to Java's bootstrap libraries, and more specifically, what classes belong to "java.net" package

With this metadata, it is possible to query for:

- Specific intra-method sites e.g. every time a variable is assigned, every time a field is accessed (see Figure 19)
- All inter-method sites e.g. every time data is passed outside a method
 - Specific inter-method sites e.g. every time data is passed to "System.out.println()" (see Figure 21)
- All inter-application sites e.g. every time data is passed outside the application classes

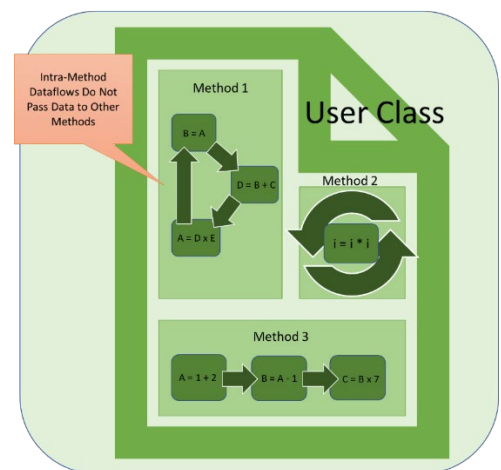


Figure 19: Intra-method

- Specific inter-application sites e.g. every time “UserClass1” passes data outside the application

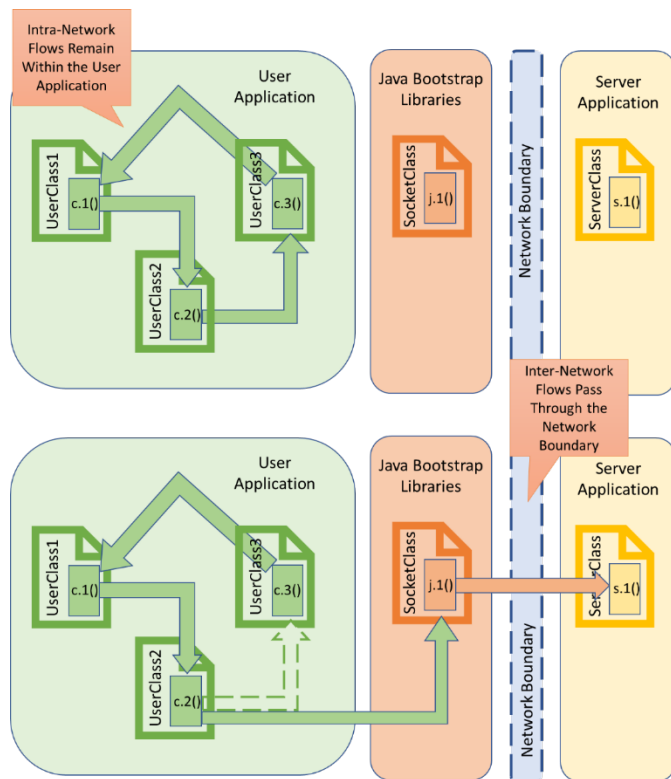


Figure 20: Intra- and Inter-network

- Specific intra-application sites e.g. every time “UserClass1” passes data to “UserClass2”
- All inter-network sites e.g. every time the data is transmitted across the network (see Figure 20)
- Specific inter-network sites e.g. all data being transmitted across the network using “java.nio.channels.SocketChannel” class

In contrast, without such metadata, we could query by method names, class names etc., only and as a result, lose the ability to distinguish data flows and code locations at a finer granularity.

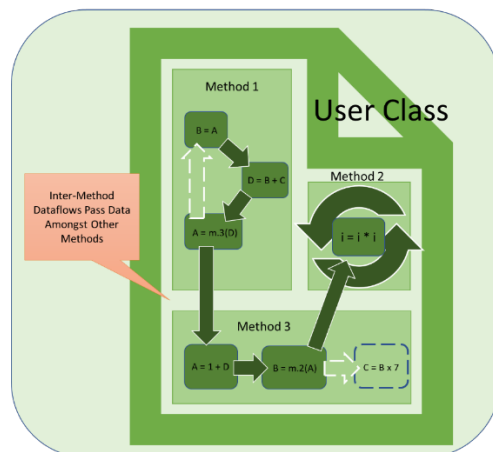


Figure 21: Inter-method

4.2.2.3.1.3 Data Frame Analysis

The analyses described so far generated metadata about data flow and code location, but not much about the semantics of the code – how traversing through code blocks changes the semantics or properties of the data. This information is needed to handle requirements on the properties of data handled by the application, specifically for this CP ensuring confidentiality of data transmitted from the client to the server.

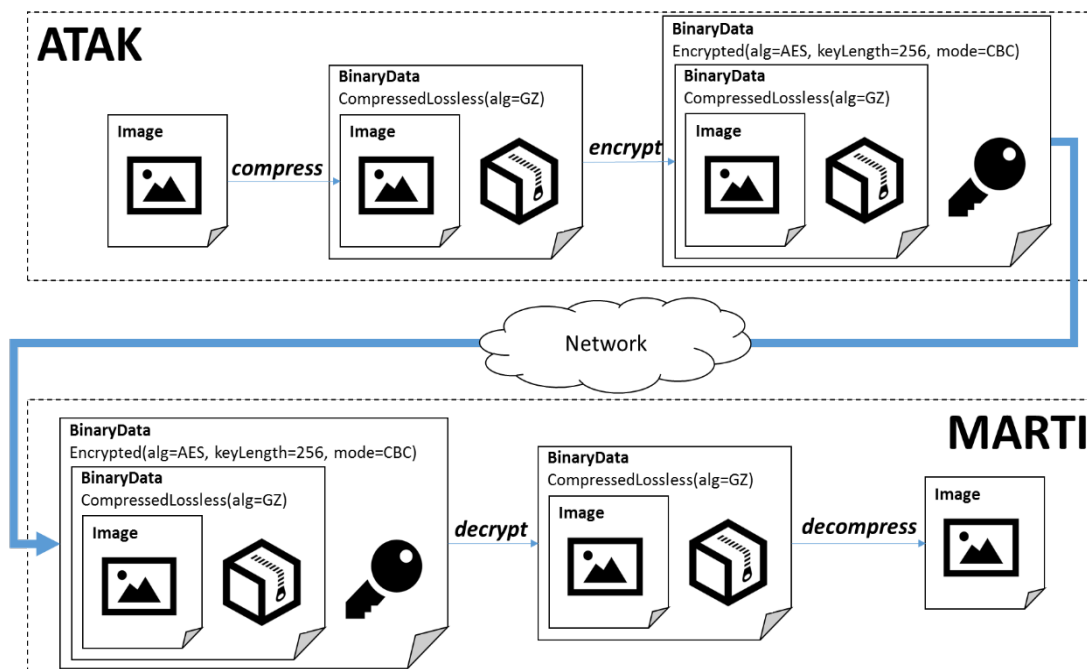


Figure 22: Illustration of data frame analysis through compressor and encryption DFUs

We have modeled code units as having property-modifying side effects on the dataflows with which their functional aspects intersect. For example, the *compress* aspect of a *compressor* code unit adds the *compressed* property to a dataflow whereas the *deflate* aspect of that code unit removes that property. By replaying dynamic call traces and following data through code regions annotated with these semantic bindings, we can perform data frame analysis. An *analysis frame* (as defined for purposes of this discussion) is a recursive structure that carries an abstract datatype and properties assumed to hold for the data unit (e.g., SA data, messages sent from client to server etc). For a given dataflow edge into a code block, it is therefore possible to record the semantic impact of that code on that dataflow in the outer layer of the frame (e.g., data coming into the compressor DFU through the compress method will add the compressed property). The next edge traversed will similarly add new property (ies) as outermost layer(s) of the analysis frame in a nested manner or removing the properties from the outermost layer(s). See Figure 22 as an illustration. A critical aspect of analysis frames is that they can be checked for consistency. For example, if a code unit expresses the expectation of receiving a frame signifying `LosslessCompressed(alg=bz2)` data value but instead receives a frame signifying

LosslessCompressed(alg=lzma), this is an inconsistency that must be reconciled for correct program operation.

Given this data frame analysis, it is thus possible to determine if the desired properties of data being transmitted across network boundary are satisfied by the application code or not. With DFUs/code units annotated with their semantic impact on dataflow, it is therefore also possible to identify which DFU/code unit needs to be spliced in the dataflow to achieve the desired property.

4.2.2.3.2 Adaptation Modules

As shown in Figure 8, the newly added confidentiality requirement is submitted to the DAS, which is processed through the CM and VM. The custom VM processing for this CP involves checking the properties of a known test message being sent from the client to the server. In the baseline application, the client sends the message in plaintext, which violates the confidentiality property. This triggers the adaptation process. Note that before the new requirement is submitted to the DAS, the application ingestion process (Figure 7) has already invoked the generic and CP-specific KBs configured in the DAS, which has inferred information about the application like the following:

- Identified the cut points in the client and server code that touches the network
- Identified that XML messages as the *data in motion* since they are transmitted across network through the pre-specified cut points
- Identified the functions involved in the data flow through these cut points—and none of them belongs DFU (user code or library) known to add encryption (decryption).

The analysis and repair techniques packaged as the AM for this configuration of the DAS then attempts to address the confidentiality violation by adding the confidentiality property to the XML messages. It uses encoded domain knowledge that messages with the `encrypted` property are `confidential` and identifies various annotated `cipher` implementations in its global code repository that add the `encrypted` property to binary data.

Inserting the cipher processing to the dataflow is realized using a proxy pattern. During repair, a proxy adaptation surface for the Android component that extends `SocketChannel` and similarly synthesizes an adaptation surface for the server that extends `Socket` is constructed. A `cipher` functionality that matches the specified requirement (essentially, calls to libraries annotated with metadata indicating that they provide the required level of encryption and decryption) is woven into the two adaptation surfaces—with the result that bytes written into the proxy `SocketChannel` will be encrypted over the wire and bytes read from a connected `Socket` will be decrypted before reaching the server’s business logic. Minor modification to the declarations make the applications use the appropriate proxy instead of the original `SocketChannel` and `Socket` objects. The configuration of the cipher is handled as a constraint optimization problem given the desired capabilities (the “best practice” specification associated with the “confidential” property).

The proxy and Application Programming Interface (API) weaving based repair mechanism is packaged as an AM, and the constraint optimization for deriving the cipher configuration as another. These are the two AMs responsible for adapting the client and server side code in this CP.

4.2.2.3.2.1 Code Repair

Proxying is an established technique for modifying those aspects of functionality that transcend a single implementation. For example, one might use proxying to enact some idiomatic behavior such as appending a magic sequence of bytes whenever writing data to a file or sending data across network via a socket. Many Aspect Oriented Programming frameworks (most notably AspectJ and SpringAOP for Java) make extensive use of proxy code units for this reason. Thus proxying is a natural choice for realizing the AM for this CP. Proxying provided a number of additional benefits as well:

- Proxying is selective enough to target specific use cases at a finer granularity and at lower cost than middleware-based solutions. For example, adapting how bytes are written to a Java `OutputStream` could theoretically be achieved (with significant difficulty) by modifying various core Java classes and native libraries instead of creating custom proxy that adds custom behavior before and after calling the function to write bytes to the delegate encapsulated in the proxy. However such changes will then be forced upon all users of `OutputStream`, which may break the contract/assumptions they have.
- Proxying requires minimal modification to existing application source code, and the modifications that are required can be easily supported by a lightweight instrumentation toolchain. Changing the declaration of the original object to the declaration of the proxy that uses the original object as the delegate could be readily automated and enacted either pre- or post-compilation (as source or bytecode modifications, respectively).
- Proxying supports adaptation use cases for both legacy and actively developed software systems. Adaptations for legacy systems can be achieved through bytecode-level instrumentation. Adaptations for actively developed software system can be based upon the synthesis of source code artifacts that can be merged into the software ecosystem using an Integrated Development Environment (IDE) or build plugin.
- Proxying cleanly decouples what things are done (the Application Programming Interface or API exposed by the proxy) from how those things are done (via calls to a delegate whose business logic is hidden and may be immutable). A proxy therefore presents a clean adaptation surface into which application-transforming functionality modifications can be injected along the dataflow edges connecting the proxy's external API to the delegate's API.

At the same time, proxying has several limitations that must be kept in mind:

- Proxying is an unwieldy mechanism for modifying implementation-specific business logic. This logic is typically protected by restrictive Object Oriented Programming (OOP) idioms such as access specifiers (`private`, `protected`) or modifiers (`final`) and therefore cannot be easily extended. While workarounds are possible even with these restrictions in place, their presence necessitates the use of extensive OOP-breaking techniques such as down-casting or reflection. Proxy-based transformations to business logic are therefore more fragile and invasive, especially for programmer-in-the-loop transformations.

- Proxying is difficult to enforce in applications where the functionality to be proxied is invoked using reflection or via dynamic call site binding (e.g., Java's INVOKEDYNAMIC instruction). Luckily, such patterns are uncommon in non-obfuscated code.
- More fundamentally, modifications at the business logic granularity are implementation-specific and therefore more usefully achieved via techniques that focus on individual code units. Proxying can be thought of as a shotgun technique, whereas some adaptations require careful surgery to be effective.

There are a number of requirements for fully automated program transformations via proxying:

1. Automated proxy-based transformations are realistic for situations where the lifecycle of a DFU can be cleanly woven into the lifecycle of a Java code unit whose functionality is to be proxied. This weaving process must be well understood before enacting the transformation.
2. Automated proxy-based transformations are realistic for situations where the code types of a transformative DFU and its injection point are closely aligned. A DFU that expects the `byte[]` type as an argument would require adaptation to work with the `ByteBuffer` type.
3. Automated proxy-based transformations are realistic for situations where the usage patterns of functionality to be proxied align with transformative DFUs. For example, a DFU with a non-blocking, asynchronous API could not be used directly in line with an `OutputStream` instance because `OutputStream` has a blocking and synchronous API (bridge logic that converts asynchronous calls to synchronous calls would be needed). As another example, a DFU for compressing an array of bytes all at once would not be easily used in a proxy for the `OutputStream` type because `OutputStream` operates on streams of data.
4. To successfully apply proxying as an automated adaptation technique, careful data and control flow

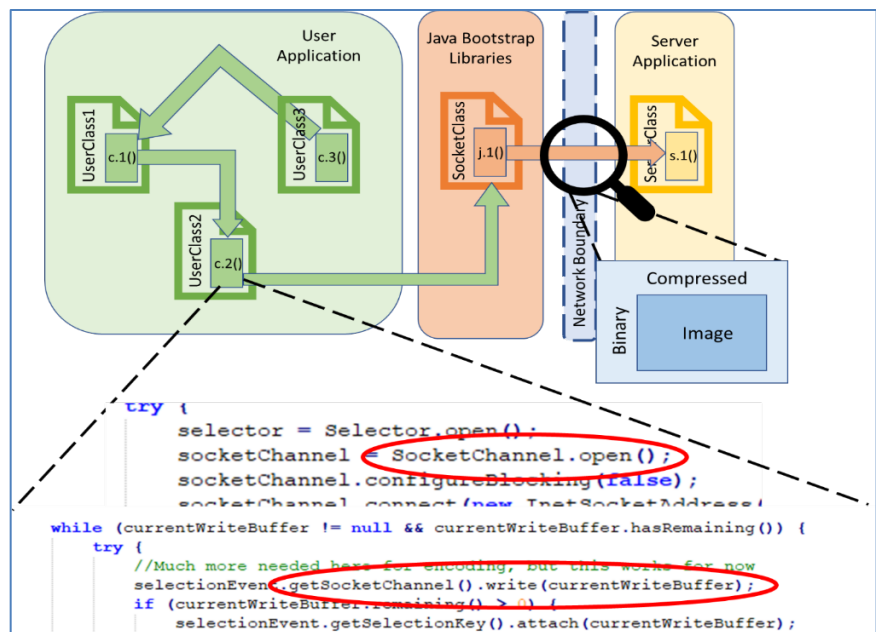


Figure 23: Dataflow (top) and actual code indicating the object and invocation to be proxied

analysis of the application must be performed sufficient to understand the points of use of the functionality to be proxied.

The AM responsible for proxy and API weaving based repair needs the details of the object to proxy and the calls to wrap and delegate, as well as the functionality to add in order to construct the proxy. In the present case, the dataflow details provide the object and invocation information (e.g., `SocketChannel` in Figure 23 for the client code), a search over the annotated code repository identifies a suitable cipher implementation (e.g., `BouncyCastle`), and constraint solving using the resource DSL (described in the next section) provides the appropriate configuration details that can be used to initialize the cipher. With this information, it is easy to construct the Java class (i.e., `CipherImplBouncycastleCrypto`) realizing the proxy.

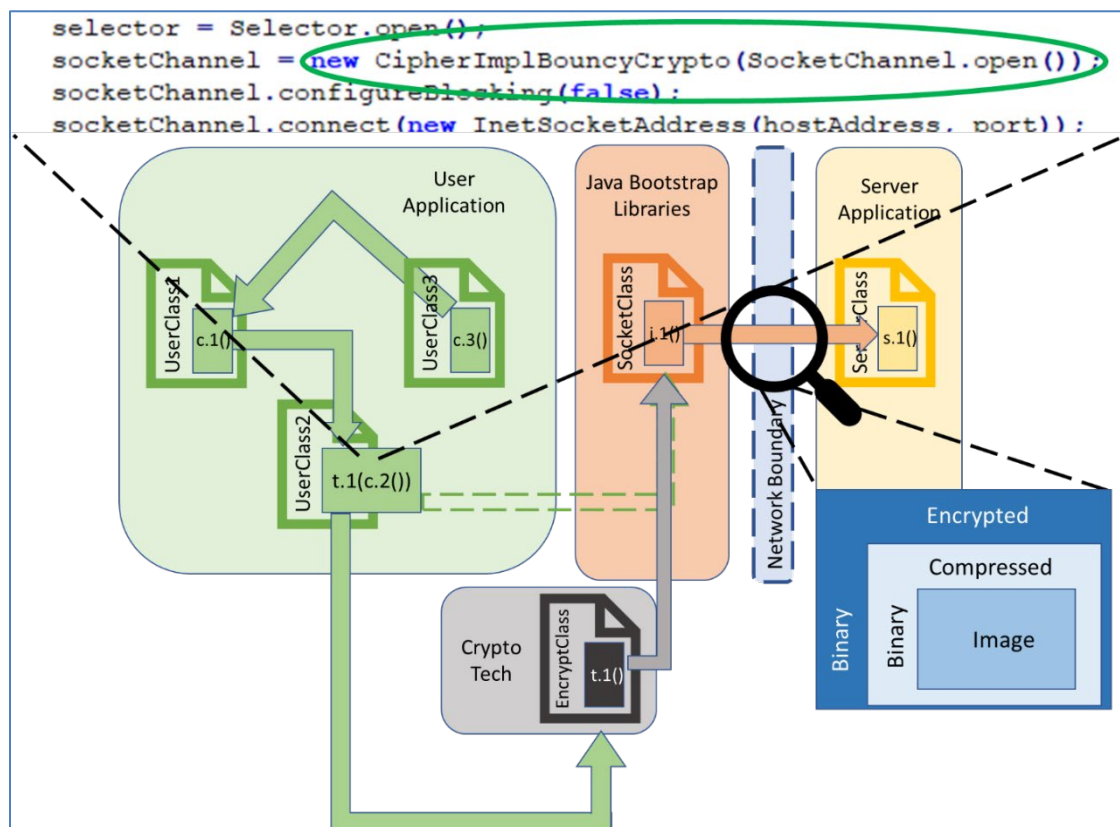


Figure 24: Code and schematic representation of inserting the proxy

Once the proxy class is constructed, weaving the proxy in the code involves simple manipulation of code as shown in Figure 24, where the initialization of the `SocketChannel` class is first passed to the newly created cipher proxy class. The result is every time the user's application uses their `SocketChannel` instance, instead of passing the data directly to the low-level Java network classes, it gets redirected through the `BouncyCastle`. Figure 23 and Figure 24 describes the client side code, the server side repair is similar,

It should be noted that before the user can compile and execute their repaired application, they will have to import the new dependencies utilized for the repair, assuming they aren't already included in the application. Although the DAS captures various dependencies, including the newly introduced ones in the knowledge base, the DAS requires a small amount of manual intervention to adapt the build configuration (as opposed to modifying the build file automatically) keeping consistent with our programmer on the loop theme.

4.2.2.3.2.2 Resource DSL

The Resource DSL enables modeling the resource requirements and provisions of many configurations of a software system. In practice, the space of potential configurations can be huge since each independent configuration option multiplies the configuration space by the number of values that option can take on. For example, a system with 10 independent binary configuration options will have 2^{10} possible configurations.

For this CP, resource DSL is typically tasked to find a configuration for a cipher that, for example, includes the following parameters: {key length, block size, chaining mode, algorithm, initialization vector} that conforms to symmetric cryptography recommendations of FIPS 140-2. The DSL-based module accepts as inputs (1) an encoding of the configuration parameters; (2) an encoding of the constraints that must be met; and (3) a configuration for a search window (since exhaustive search is intractable). It then produces as output a bag of ranked viable configurations discovered within the limits of the search window. The best configuration is then into the appropriate application code unit(s). If no viable configurations are found, the DAS aborts the repair process and issues a warning that no viable configurations were discovered.

In Phase 1, DSL models were used only to validate the configurations recommended by the SQRQL component of the DAS. To support this, the semantics of the Resource DSL was defined by a two step process: (1) configure the model, (2) evaluate the model against a given initial resource environment and final mission requirements. At the end of these two steps, either a final resource environment is produced if the configuration is successfully validated, or else an error is produced indicating the resource constraint or requirement that was violated.

In Phase 2, the semantics of the resource DSL was extended [40] to support validating many or potentially even all of the configurations of a model at once using a technique called *variational execution* [24, 25, 26, 41]. This extension changed the interaction model of the semantics into a three step process: (1) partially configure the resource model by assigning values to a (potentially empty) subset of the configuration options, (2) evaluate all remaining configurations of the model using variational execution against the initial (also potentially variational) resource environment and mission requirements, (3) query the results to determine the success or failure of individual configurations, or to return suggested configurations given some additional constraints.

Unlike the non-variational Phase 1 semantics, a violated resource constraint or requirement should not immediately return an error since other configurations may still be valid. In fact, for any non-trivial system, we would expect a mix of valid and invalid configurations. Therefore, the output of the semantics is changed from *either* a resource environment or an error to *both* a variational resource environment that captures the final resource state of all successful

configurations and a variational error that captures violations of all failed configurations. These variational results can then be queried (in step (3)) to determine which configurations succeeded (and their corresponding remaining and provided resources) and which configurations failed (and why). The variational execution in step (2) is the expensive phase of the semantics, while the results can be reused for many different efficient queries in step (3).

Since the configuration space of a resource model can be huge, variational execution only works for highly configurable models if there is a substantial amount of sharing among different configurations. In previous work, PI Walkingshaw and collaborators have had significant success applying the techniques of variational execution to program analyses that operate at the level of types. For example, we have defined variational typing [44, 45] to automatically infer and check types for many variants of a program at once and used this technique to build speculative program analyses (analyses that efficiently explore huge numbers of hypothetical changes to programs) that address problems in error location [47], validating C++ templates [46], inferring types for generalized algebraic data types [48], and improving the type coverage and performance of programs written using gradual types [49, 50]. All of these applications work by performing a *complete search* of an exponential number of hypothetical program variants. Efficiently searching an exponential space is possible at the level of types because many program variants have the same type, and so a significant amount of the search can be directly reused among variants. Since many resources are modeled as (hierarchies of) numeric values, where every small change in the configuration will have an effect on the value of the resource, one of the major research questions for work on the Resource DSL is how well these techniques would scale to this less abstract domain.

To address the cross-app configuration scenario posed by CP2, we first had to identify the relevant “resources” in the scenario. We started by partitioning the resource environment into two namespaces, one corresponding to the *Server* application and one to the *Client* application. To facilitate coordination of the encryption DFUs in these two applications, we modeled the relevant encryption provisions as a 4-tuple:

- *Algorithm* captures the cipher or encryption algorithm used. Our scenario provided for 29 possible algorithms: *AES*, *ARIA*, *Blowfish*, *Camellia*, *CAST5*, *CAST6*, *DES*, *DESede*, *DSTU7624*, *GCM*, *GOST28147*, *IDEA*, *Noekeon*, *RC2*, *RC5*, *RC5_64*, *RC6*, *Rijndael*, *SEED*, *SEEDWrap*, *Serpent_128*, *Skipjack*, *SM4*, *TEA*, *Threefish_256*, *Threefish_512*, *Threefish_1024*, *Twofish*, and *XTEA*.
- *Mode* indicates the encryption mode used, for example, determining whether/how the message is broken down into smaller parts. There are 9 possibilities: *ECB*, *CBC*, *CTR*, *CFB*, *CTS*, *OFB*, *OpenPGPCFB*, *PGPCFBBlock*, and *SICBlock*.
- *Padding* indicates the padding scheme used to prepare a message for encryption. Our model of the system included 8 padding schemes: *ZeroBytePadding*, *PKCS5Padding*, *PKCS7Padding*, *ISO10126_2Padding*, *ISO7816_4Padding*, *TBCPadding*, *X923Padding*, and *NoPadding*.
- *KeySize* indicates the number of bits used in the encryption key. Our model includes 8 possible key sizes: 8, 16, 24, 32, 40, 48, 56, and 64 bits.

Thus the encryption algorithm used by the server's encryption DFU would appear in the global resource environment as */Server/Cipher/Algorithm*, while the padding scheme used by the client would appear as */Client/Cipher/Padding*.

Not all possible configurations are valid since, for example, not all algorithms support all key sizes. Given a list of compatibility relationships, we programmatically generated a set of resource constraints that ensure the chosen algorithm is compatible with the configuration of the other options. Generating these constraints was a one-time step. The constraints then became a static part of the application model.

Additionally, some configurations have external resource constraints. For example, a global constraint is that larger key sizes (24 bits or greater) require the presence of a corresponding *StrongEncryptionPolicy* on the server or client machine. The presence of this resource was modeled by a unit value at */Server/StrongEncryptionPolicy* or */Client/StrongEncryptionPolicy*, respectively.

Finally, we modeled the two different encryption provider DFUs, *Javax* and *BouncyCastle*. The *Javax* DFU provides a limited subset of algorithms, modes, and paddings. The *BouncyCastle* DFU provides all valid configurations (subject to the global constraints above), but has an additional external resource constraint: if the *AES* encryption algorithm is selected, *BouncyCastle* requires corresponding hardware support for the *AES-NI* instruction set. This is modeled similarly to the strong encryption policy constraint described above.

The application model describes all possible configurations of both the client and the server by modeling the scenario as two independent choices between the *Javax* and *BouncyCastle* DFUs. Additionally, to support efficient querying of different hardware scenarios, we can model the initial resource environment as a *variational environment* where we effectively defer environmental configuration decisions, such as whether the server or client have strong encryption policies or AES-NI instruction set support.

With the scenario modeled in this way, we can perform variational execution (step (2) of the semantics) on the fully unconfigured variational application model and the variational initial resource environment to produce outputs that efficiently encode all possible configuration scenarios.

4.2.2.4 Evaluation

For simplicity, we paired ciphers and key sizes together to define encryption levels and used CP formulation (encryption requirements, cryptographic resources available at client and server sides) mirroring the requirements of some known standards when known as shown in Table 9.

Table 9: Sample CP configuration

Encryption Requirement	AES-128, AES-256, DES-56, Rijndael (any key size)
Client Resources	Hardware AES, Strong Crypto
Server Resources	Hardware AES, Strong Crypto

The selectable resources on the client and server were simulated since they were running in a virtualized environment that limited our control over the actual hardware resources. These

resources were Hardware Accelerated AES by the CPU and support for Strong Crypto which Java 8 does not support by default due to export restrictions.

4.2.2.4.1 Internal Evaluation

We performed a number of basic evaluations of the encryption requirements with a non-exhaustive list of resource combinations. Following the adaptation, re-execution of the tests concluded that the data could no longer be sent or received by plaintext clients but could be sent and received by adapted clients. Manual examination of the changes indicated that the proper ciphers and key sizes were being used by the adapted client and server applications. Although there were some issues uncovered in our data used to seed the Resource DSL and usage of incorrect parameters for some ciphers, after resolving those issues, everything validated and chose the alternative BouncyCastle library when the native library did not have any ciphers or key sizes that met the encryption requirements with the exception of when the resources were not adequate to perform an adaptation.

4.2.2.4.2 External Evaluation by the Voice of the Offense Team (Lincoln Laboratory)

The result of the repair workflow is an adapted application with the same apparent functionality as the original but modified to operate under a degraded or altered execution environment problematic for the original software. A retrospective `diff` of the repaired application would reveal a small number of modifications to the points of use of problematic code units (with the instantiation of a proxy used in place of the original problematic code unit), the addition of the proxies themselves, and finally the inclusion of additional classes not present in the original application but referenced within the proxies.

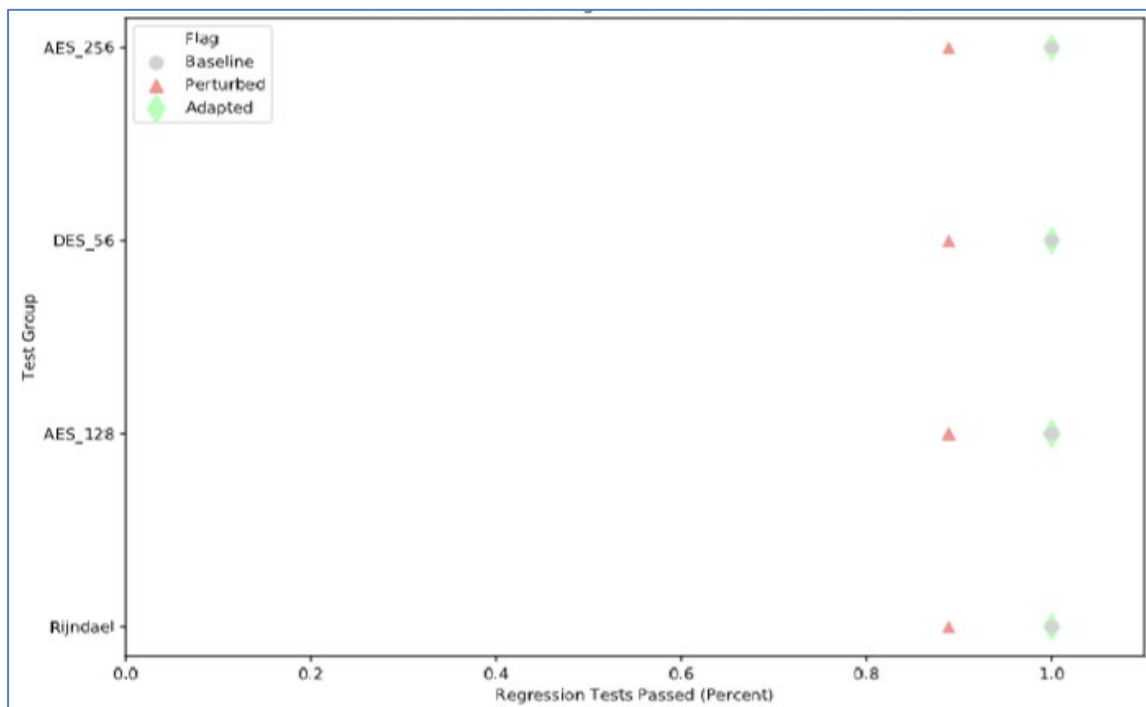


Figure 25: Regression tests passed per test group for Phase 2 CP2

A summary of the results obtained during evaluation by an independent validation team from MIT Lincoln Laboratory are shown below in Figure 25 and Figure 26. Figure 26 shows a degradation in functionality (which manifests as failing test cases) across the algorithm and configuration space in the perturbed but not adapted system. The figure also shows that the repaired application operates as expected with no test failures.

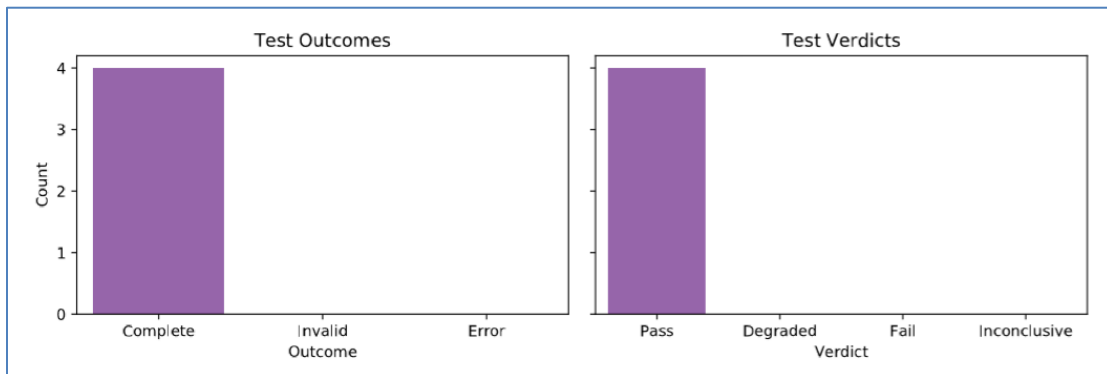


Figure 26: Evaluation summary for CP2

4.3 Handling Changes in Deployment Environment

4.3.1 Adapting to Resource Constrained Deployment Environment (Phase 1, CP2)

4.3.1.1 Motivation

Even if requirements remain the same, changes in the operating environment can cause the application to underperform, or even fail to meet requirement. Software adaptation is often needed when complex constraints, like a server's bandwidth usage, often exceeds what is available and domain knowledge is required to apply remediation strategies that extend beyond component substitution. The second CP in Phase 1 demonstrated IMMORTALS capabilities addressing adaptive pressure caused by constrained resources in the deployment environment.

4.3.1.2 CP Description

This CP uses the same tactical SA application as Phase 1 CP1, consisting of ATAKLite clients interacting with the server over the network, but now the deployment environment is bandwidth restricted, and the SA data we care about is image update. It is important the clients are able to share images in a timely manner.

As shown in Figure 9, variation points may be associated with a resource consumption profile that helps the DAS system evaluate the relationship between the variation point and a resource type, such as network bandwidth, CPU and memory usage. As the DAS iterates each feature requirement in the deployment model and their linked variation points, it verifies whether the resource consumption profiles linked to each variation point are satisfied given the requirements and resource provisions in the deployment model, such as the number of clients that need to be supported and the bandwidth available to the server. For this CP, the resource consumption profile focused on total consumed bandwidth at the server as a function of the following factors:

1. Number of clients
2. Size of a Cursor-on-Target XML message
3. Size of an image reported by the ATAKLite device camera
4. The required minimum reporting rate for basic position information
5. The required minimum reporting rate for images captured by field devices

The server consumes bandwidth from clients sending CoT messages and images to the server as well as the server sharing this information from every client to every other client. The general formula is:

$$NumClients * Rate * MsgSize + NumClients * (NumClients - 1) * Rate * MsgSize \quad Eq: 1$$

This formula is evaluated where *Rate*=*Minimum Position Location Report Rate* and *MsgSize*=*Average Cursor-on-Target Message Size* (400 bytes). This formula is evaluated similarly for images, using *Minimum Image Report Rate* and *Image Size*. The two results are added to compute the total consumed bandwidth at the server. The resulting value is compared to the total allocated for the mission (as defined within the deployment model) and if a violation is detected, the DAS follows a series of domain-specific steps to remediate the issue.

4.3.1.3 Relevant IMMORTALS Mechanism and Capabilities

The overall workflow for handling adaptive pressure from changes on deployment environment is as follows. The Change Manager detects from the submitted deployment model that resource violation such as bandwidth usage against availability exists at a variation point. This triggers the Adaptation Module to search through the software metadata and knowledgebase containing relevant domain knowledge to identify a remediation strategy, source code building block that realizes the strategy and source code location related to the offending variation point where such a strategy can be applied. It then performs a resource provisioning analysis it determine applicable parameters that customizes the remediation strategy, and performs source code modification. The following subsections describe the techniques in more details.

4.3.1.3.1 Knowledge Builders

The overall workflow summarized above assumes the existence of software metadata and domain knowledge. Knowledge builders produce the metadata, however, domain knowledge is typically hand coded.

Examples of software metadata include information about the application's dataflow, DFU characterization such as the amount of memory footprint reduction caused by a given scaling factor input for the ImageScaler DFU.

Table 10: DFU metadata example

Description	Value (Class Paths Omitted)
Initial Class Constructor Parameters	ConsumingPipe<Bitmap>
Adapting Class Constructor Parameters	Integer, ConsumingPipe<Bitmap>
Adapting Class New Parameters	First Parameter, ScalingFactor
Target ATAKLite Java File	SACommunicationService.java

Examples of domain knowledge include rules and guidance such as to reduce bandwidth consumption one can reduce rate of data transmitted over the network, reduce the number of concurrent producers and consumers, or reduce the memory footprint of dispatched data etc.

In Phase 1, while the KBs and the AMs were built in parallel, we took a simplification by hand creating some of the metadata (see Table 10). Later in the project, we validated that the hand coded metadata can indeed be produced by KBs.

4.3.1.3.2 Adaptation Module

4.3.1.3.2.1 Searching Software Metadata

This process begins with a target *resource type* and *action* (*network bandwidth* and *reduction*, respectively, in the current case). Simply put, the DAS submits a SPARQL query that looks for ways to reduce bandwidth usage. This first query may result in a set of additional resource type/action pairs, including reducing the memory footprint on dispatched data. Each of these pairs,

in turn, form the parameters for the next query. This process iterates until a functionality label or URI (`#ImageResizer`) is found. This functionality URI is used to query the DFU repository for an available implementation. One complication with the above simple scheme is that remediation strategies may have refinements or constraints on which types of data they apply to. For example, one chain of reasoning that results from the above iterative process is shown in Figure 27.

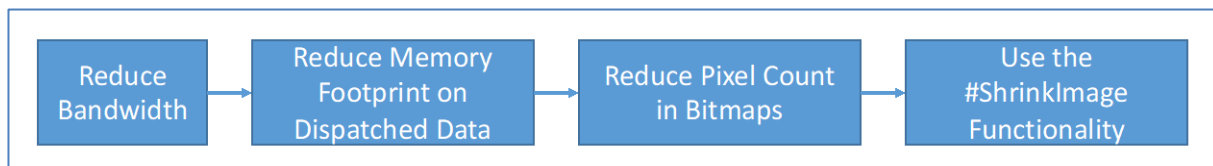


Figure 27: Property Action Chain for Bandwidth Reduction

The *Reduce Memory Footprint on Dispatched Data* strategy will likely branch out to multiple distinct strategies depending on the application data type. To determine which strategy applies to the current variation point, the DAS needs to perform a dataflow analysis beginning at the variation point and ending at a network dispatcher. Initially the dataflow information was hand-coded into the RDF store. Later on, we verified that dataflow analysis performed at build time produces the kind of data we expect, and the resulting dataflow can be encoded as an RDF graph and linked to the variation point (as shown earlier in Figure 9). The dataflow analysis is used to verify that the `#ImageResizer` functionality is appropriate for the current dataflow, which reveals a common code block that embeds images from the device camera into the CoT message dispatched to the server. If the DAS can insert a DFU which implements the `#ImageResizer` functionality into the correct location on the ATAKLite client, it can reduce bandwidth consumption at the server. The details of how the source code is transformed are described later in this section.

The ImageResizer DFU accepts both a Bitmap instance and a scaling factor in constructor. The choice of scaling factor obviously impacts the extent of bandwidth reduction that is achieved. Moreover, a correct value depends on multiple factors, including the number of clients and the frequency of reports which contain images. Excessive scaling will negatively impact the usefulness of images; hence, an optimal value is desirable. The DAS invokes a resource DSL program that considers these factors and produces an optimal scaling factor.

4.3.1.3.2.2 Resource DSL for Parameter Provisioning

We used the Resource DSL to implement and maintain a model of the relevant dependencies, requirements, and capabilities of the system. The model of the system essentially does the following:

1. Loads and configures a model of an ImageProducer DFU that models the network bandwidth consumption of images generated and transmitted by the system, and which contains as a component within it the ImageResizer DFU described above.

2. Loads a simple model of a SituationalAwarenessProducer DFU that models the network bandwidth consumption of situational awareness data generated and transmitted by the system.
3. Checks that the bandwidth consumed by the system is not greater than the bandwidth available within the system.

Steps 1 and 2 of this model are part of the application model, while step 3 is validated in the mission requirements.

The ImageProducer DFU is configurable by six parameters. The numeric values *x-axis resolution*, *y-axis resolution*, *scaling factor*, and the boolean value *color* describes relevant properties of the images; the numeric value *image rate* describes how many images are transmitted each minute; and the boolean value *compressed* indicates whether the image will be compressed. From these parameters, an average *image size* can be computed in bytes using the *resolution* and *color* parameters, possibly reduced by the *scaling factor* and the average reduction factor of compression if images are *compressed*.

Although the modeled system is quite small, we demonstrate the hierarchical modeling capability of the Resource DSL by decomposing the ImageProducer DFU into sub-DFUs ImageResizer, ImageScaler, and ImageCompressor, which model the various parts of the image production process.

The SituationalAwarenessProducer is configurable by just one parameter: *PLI rate* indicating the rate that position location information is transmitted.

Naturally, larger image sizes, higher image rates, and higher PLI rates all increase network bandwidth consumption. The Resource DSL ensures that for a particular configuration of the system, the maximum network bandwidth is not exceeded.

In this phase, models in the Resource DSL were used only to validate a configuration of the system against the initial resource environment and the mission requirements (in later phases, the Resource DSL models are used to directly solve for optimal configurations). Thus, the DAS would send several “queries” to the Resource DSL interpreter in the form of different values of the configuration options of the system, and the interpreter would respond with a Boolean result indicating whether the configuration successfully validated. Through repeated queries, the DAS iteratively searched for an optimal scaling factor.

Once an optimal scaling factor is determined, the DAS component begins the task of modifying the source code so the the ImageResizer is initialized and used at the correct location in the code.

4.3.1.3.2.3 Source Composition

This capability relies on the product line architecture of the ATAK-lite Android application. The ATAK-lite application uses Java objects, which are instances of class-level DFUs, organized in a pipeline pattern to handle staged processing of SA data such as images and location inform from sources (i.e., camera or location providers) to destination (i.e., storage or to the network to be handed to the TAKServer). It is possible for a pipeline to end in a sink, i.e., there is no next stage—the last stage *consumes* the data.

The realization of the pipeline utilized the *ConsumingPipe* interface that takes entire object

```
1 public interface ConsumingPipe<InputType> {
2     public void consume(InputType input);
3 }
4
```

Figure 28: ConsumingPipe Interface

provided as a generic parameter (see Figure 28). The *consume* method will perform the operation on the object, and if it is an endpoint of the pipeline, stop there. If the *ConsumingPipe* is expected to pass the result on

to another step in the pipeline, the constructor of the *ConsumingPipe* will take a *ConsumingPipe* parameter with a generic type matching what is produced by the *consume* method. The *consume* method of that parameter will in turn be called at the end of the parent objects *consume* method, propagating the modification to the next portion of the pipeline.

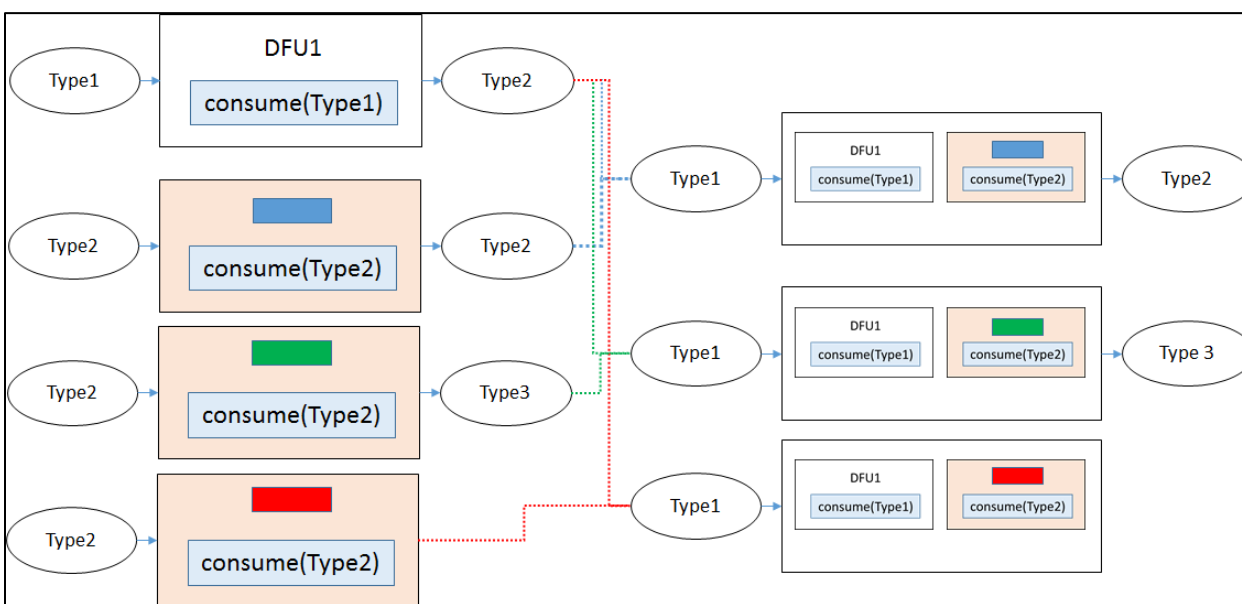


Figure 29: Illustration of DFU Composition

If DFUs conform to the pipeline pattern and the *ConsumingPipe* interface, it is possible to compose new DFUs by combining multiple DFUs. This is illustrated in Figure 29, where we show three different types of DFUs that are compatible with DFU1 for composition on the left, and on the right we show the resulting composed DFUs. Each the DFU that gets composed in (i.e., the beige boxes in Figure 29) can have additional parameters to their function (constructor, or the main work function that executes its business logic)- these are described as part of the software metadata associated them, and values need to be provided by the AM that invokes the source composition mechanism. In the current CP, the resource DSL provided the values used by the source composer.

The composition approach described above minimized the complexity of code adaptation, and allowed reusability of code components (DFUs and metadata). This approach also allowed us to update the functionality by simply adding the newly created package to the build system dependency manifest and substituting instances of the old class in the target document with the new class.

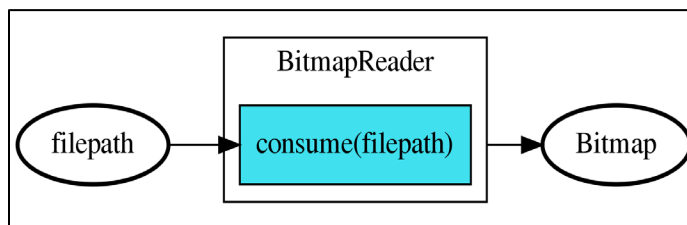


Figure 30: Schematic of the BitmapReader DFU

In this CP, source composition was used to reduce the bandwidth used by the ATAKLite client by composing a bitmap reader with an image scaler which provided the capability of a scaled bit map reader (BitmapReaderScaler DFU). The original BitmapReader and composed DFUs are shown schematically in Figure 30 and Figure 31.

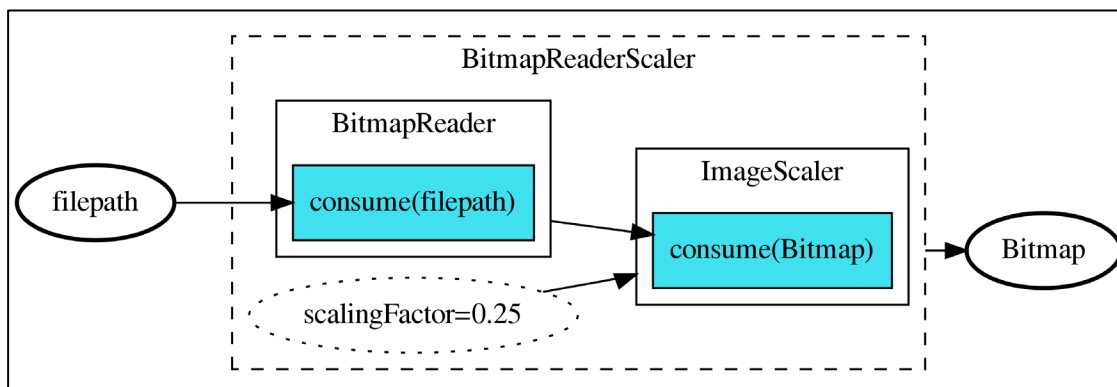


Figure 31: Schematic of the composed DFU

The resulting composed Java class is shown in Figure 32 (the class paths and uniqueness identifiers have stripped for readability). The initial *BitmapReader* DFU is declared as the "head" field in the new DFU and instantiated in the constructor. Within that, a new instance of the adaptation DFU *BitmapScaler* is created with the new parameter and connected to the output of the *BitmapReaderScaler*.

The project structure and build file are then created using a generic template, substituting unique identifiers where necessary and adding the initial and adaptation DFU packages to its dependencies. It is then automatically built and made available to the target application, ATAKLite. The new DFU dependency is then automatically injected into ATAKLite's dependency list, previous instances of *BitmapReader* in *SACommunicationService.java* are

automatically replaced with *BitmapReaderScaler*, the application is rebuilt, and adaptation is complete.

```
1  BitmapReader head;
2
3  public BitmapReaderScaler< > {
4      public BitmapReaderScaler(ConsumingPipe<Bitmap> tail) {
5          head = BitmapReader(
6              new BitmapScaler(
7                  0.25,
8                  tail))
9      }
10
11     public consume(String filepath) {
12         head.consume(filepath);
13     }
14 }
```

Figure 32: Simplified code listing of the composed DFU

4.3.1.4 Evaluation

4.3.1.4.1 Internal Evaluation

A significant part of our internal evaluation was focused on mitigating a variety of emulator issues and benchmarking where the delays were during start and trying various ways to mitigate them. Ultimately, we had to stagger launching them to avoid unexpected freezes and found launching more than 4 emulators to be infeasible due to the significant increase in startup time for each emulator. We in turn ran tests with a variety of value combinations and validated the bandwidth.

It generally behaved as expected. However, due to underestimating the size of the base CoT message without an image, we found it was possible to violate the bandwidth unexpectedly with some select settings.

4.3.1.4.2 External Evaluation by the Voice of the Offense Team (Lincoln Laboratory)

The challenge problem was evaluated by MIT Lincoln Laboratories using the text procedure described in this section. The platform under test is unchanged from CP 1. The CP 2 perturbation parameters are shown in Table 11.

As with Phase 1 CP1, Phase 2 CP2 was tested using the same 3 stage approach shown in Table 7.

Table 11: CP 2 Perturbation Parameters

Parameter	Possible Values
Bandwidth (Total Server Bandwidth)	1 Kbps – 10 Gbps
Number of Clients	2 – 6
Maximum Time Between Situation Awareness Reports	1k – 60k ms

The focus for evaluation was to determine how Baseline B and the Challenge stage responded as the parameters in Table 11 were changed. Tests 1-4 and test 11 in Table 6 were executed and required to pass for each test. The IMMORTALS system returned one of two values for each test: (1) Solution Possible (given either the baseline application with no changes or a modified application); or (2) Solution Infeasible (the resource constraints are too tight to build an application that will satisfy the mission requirements). The verdict on whether the application's intent was satisfied is measured by comparing the expected outcome of the test with the actual outcome as shown in Table 12.

Table 12: CP 2 Intent Verdict

Condition	Intent Verdict
Error reported to MIT test harness	ERROR
Expected Status = Actual Status for all tests where Expected Status ≠ Not Applicable	PASS
Expected Status ≠ Actual Status or solution is infeasible	FAIL

A total of 400 tests were evaluated where 198 tests completed successfully (Figure 33). Of the remaining, 62 were marked invalid due to the fact that the perturbation was not sufficient to cause Baseline B to fail. The remaining 202 experienced unexpected test errors. This was largely due to the fact that scaling the number of Android virtual machines for

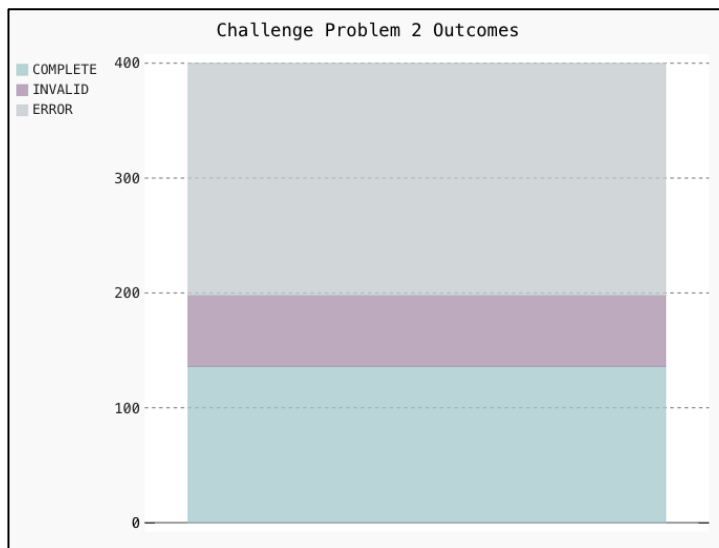


Figure 33: CP 2 Outcomes

the number of clients caused instability in the virtual machine environment. This is a challenging aspect of testing Android based applications the team avoided in future CPs.

Of the tests marked as Complete (198), 117 passed and 19 failed. This is the intent outcome and is shown in Figure 34.

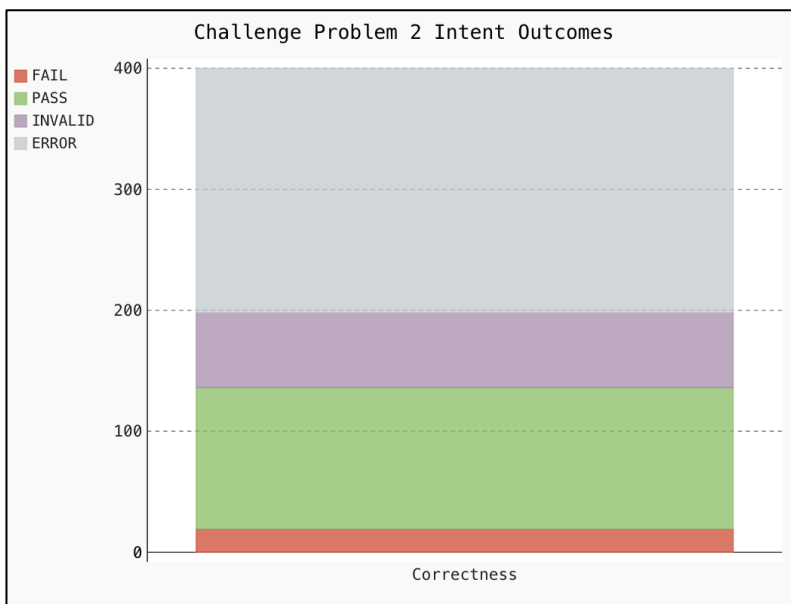


Figure 34: CP 2 Verdict Outcome

4.3.2 Adapting Flight Test Plans (Phase 3, CP1)

4.3.2.1 Motivation

Testing and certifying an aircraft is a very costly process that involves many systems and scheduled airtime to perform. In the event of a faulty DAU, swapping it for an identical DAU is the typical solution. However, this can result in a delay in testing if another identical DAU is not available locally or lead to a shortage of the specific DAU for other programs that may be able to utilize it better. A far better solution would be to search a local DAU inventory for a suitable cheaper and immediately available replacement, allowing the optimization of resources and preventing the delay of a flight test.

4.3.2.2 CP Description

4.3.2.2.1 Relevant Domain Basics

A Flight Test Configuration (FTC) is an XML document following the MDL [37] specification that indicates how a set of DAUs (or *NetworkNodes* based on the actual XML element type we are considering) can be utilized to record data on an aircraft during a flight test. For this challenge problem, we are focusing on several key elements of the specification: *Measurements*, *DataStream*s, *NetworkNodes*, *Modules*, *Devices*, and *PortMappings*. The diagram in Figure 35 displays an overview of how a FTC containing two DAUs may be structured in the context of this challenge problem.

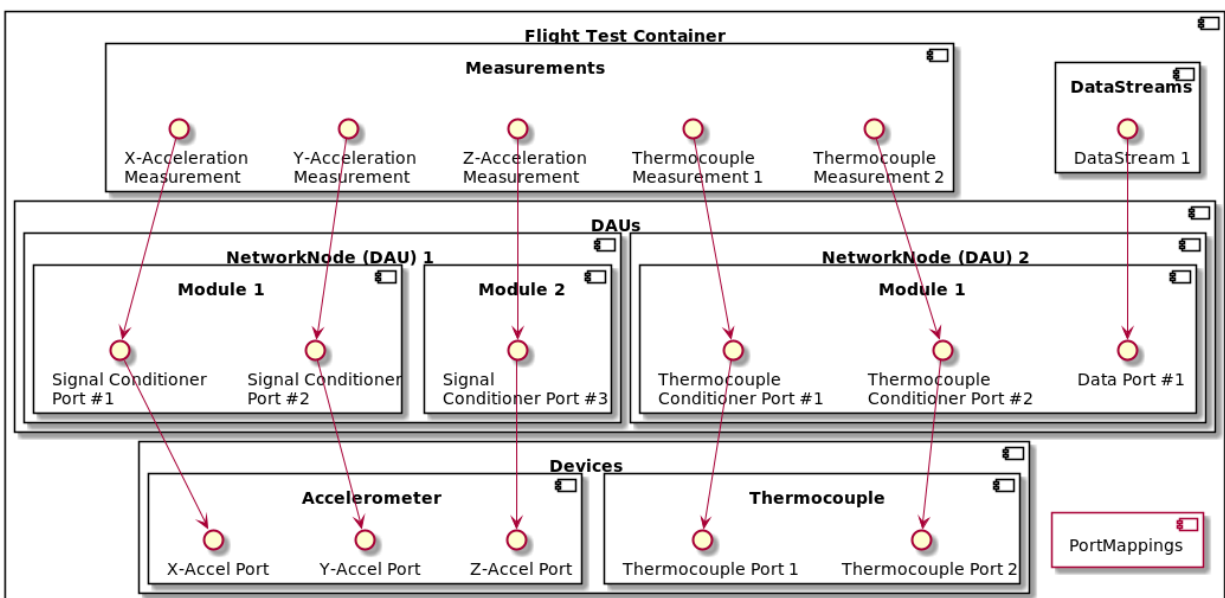


Figure 35: Sample flight test configuration

PortMappings are used to define how *Devices* connect to a *NetworkNode* through *Ports* and how the data from that DAU can be interpreted in terms of a *Measurement* or *DataStream*. Although *Measurements* and *DataStreams* do not necessarily have a physical connection as shown in the diagram, it is simplest to describe it as one in terms of adaptation and the *PortMapping*. In this diagram, each *Device* produces data that is sent from its Port that connects to a corresponding *NetworkNode Module Port*. Each of these ports then in turn often has an association to a *Measurement* to indicate how to interpret the data received. DAUs may also have their own *DataStreams* which provide data that is not directly tied to a specific port. The result is a configuration that indicates how the devices and DAUs are connected and instructions of how to interpret the data produced by the DAUs. This data is utilized by actual flight tests which were beyond the scope of the challenge problem.

The *Measurements*, *Devices*, *Device Ports*, *NetworkNodes*, and *NetworkNode Ports* have attributes that indicate additional details regarding how things must be configured in order to utilize the FTC on actual hardware. *Modules* are interconnected within a *NetworkNode* to provide specific functionality. They were considered as another potential avenue to utilize the FTCS and Resource DSL but were ignored in favor of focusing on DAUs, leaving each DAU as a discrete component in which the modules could be ignored.

The conversion of XML to OrientDB resulted in SwRI introducing a couple key concepts into OrientDB that were assumed by all parts of our system to adhere to a hierarchical data structure:

- XML Hierarchical relations were represented with a directed edge of type *Containment* from the child node to the parent node.
- XML identity and identity rederences (IDs and IDREFs) were represented with a directed edge of type *Reference* from the node that contained the IDREF to the node that contained the ID.

4.3.2.2.2 Domain Adaptation

An MDL Flight Test Configuration (FTC) is designed for a single purpose- to indicate a configuration for usage. Due to this, some customizations were necessary to provide the data necessary for adaptation since concepts such as multiple valid values or ranges were not valid MDL but were necessary to define attribute usage for the adaptation process.

Three types of information were necessary in order to execute an adaptation scenario:

1. How a DAU is used, which shall be defined *FTC DAU Usage*, *DAU Usage*, or *Usage*. This is reflected by unmodified MDL.
2. How a DAU can be used, which shall be defined as *DAU Requirements* and is relevant to the DAU Inventory.
3. How a DAU must be used, which shall be defined as *FTC Requirements*, and is relevant to the configuration containing a faulty DAU.

To help enforce the difference, consider a knife. When I am eating, a knife **is** used for *eating*. However, it **can be** used for *eating*, *hunting*, or *cooking*. However, in the cafeteria, it **must** be used for *eating* or *cooking*. Since MDL supports arbitrary elements through the *GenericParameter*

element, we were able to annotate the DAU elements with additional data without invalidating the MDL. The general concept we followed was that values and elements within a *GenericParameter* would have the same name elements in the parent object, indicating FTC Requirements in the FTC with a faulty DAU or DAU Requirements in the DAU Inventory. Then if this data is desired, the corresponding values from the *GenericParameter* can then be remapped to the parent object. It can be assumed the FTC contains the *FTC Usage* and *FTC Requirements* and the DAU Inventory contains the *DAU Requirements*.

4.3.2.3 Relevant IMMORTALS Mechanisms and Capabilities

4.3.2.3.1 Flight Test Constraint Solver (FTCS)

The Flight Test Constraint Solver was intended to act as the translator between the more complex Flight Test Configurations and the DSL. Much of the structure for reading, transforming, and validating is decoupled from the Flight Testing domain where possible. The main exceptions were when determining the *PortMapping* values and validating the constraints relative to their requirements for internal testing. In order to simplify things so that the DSL could process them, instead of requiring the DSL to have an understanding of the relationships among Measurements, DataStreams, DAUs, Modules, Devices, and PortMappings, we decided to flatten everything from the perspective of the DAUs. This allowed everything to be presented as a DAU matching DAU Usage, DAU Requirements, or FTC Requirements. The differentiating factor for these three representations is from how custom transformations are applied to each one to rewire *GenericParameters* and elements associated through the PortMappings into the existing DAU structure.

Due to our initial lack of knowledge of the complex MDL domain and the significant potential for coordination issues with SwRI, we prioritized architecting things in a way that would be flexible, help guide us to the right questions to ask, and allow us present what we understand clearly. We continuously followed the following workflow to do so as part of our ingestion process for adaptation every time they produced a new example, which caught many issues in their examples and knowledge gaps in our understanding of the examples:

1. Gather Data: For a set of root elements, gather all specified node types and their specified attributes, conserving the overall structure for each root element. These root structures were the Measurements, DataStreams, Devices, and NetworkNodes within the provided FTC and DAU Inventory. This allowed omitting values that were descriptive or not relevant to determining a replacement.
2. Gather Relations: Collect PortMapping details to determine how the tertiary roots (*Measurements*, *Devices*, and *DataStream*s) are related to the primary roots (*NetworkNodes*).
3. Transform: Trim, remap, and augment the structures following the JSON-based domain transformation instructions and relations data to align the structure of the *DAU Requirements*, *FTC Requirements*, and *FTC Usage*.

4. Validate: Raise warnings for unexpected values, dangling references, unaccounted for attributes, unaccounted for nodes, data clobbered during transformation, or anything else that is not known to us or violates constraints communicated to us from SwRI.

4.3.2.3.1.1 Data Collection and Transformation

The FTCS data collection strategy was designed to take generic hierarchical data with a format-specific data collector (which consisted of OrientDB and XML in our case), apply a JSON-based configuration for the domain, and produce a domain and format-agnostic HierarchicalData object. This object mirrored the original hierarchical structure of the input data and retained references to the original raw objects to be adapted after a solution was chosen, but only contained the node attributes defined in the JSON configuration. After the collection of data, a sequence of transformations were applied to align the FTC Requirements and DAU Requirements so they could be directly compared by the DSL. These transformations were added to and improved over time as necessary as we iterated through discussions with SwRI.

As indicated earlier, the transformation mechanism was built to be independent of the data, utilizing rules defined in a JSON file. This JSON file contained a set of general transformation rules to apply to the domain, as well as specific transformation rules for DAU Usage, FTC Requirements, and DAU Requirements. This allowed data to be trimmed for each category of DAU specifications and then transformed to align for comparison. The transformations broke down as indicated in Table 13.

Table 13: MDL to DSL transformation sequence

Operation	Description and Example
Remove Ignored Trees and Attributes	These rules defined sections that contain collected data that has been confirmed as ignorable. Generally, this includes nodes such as the Ports used internally within a DAU to connect the modules. It is also used to remove DAU Usage, DAU Requirements, and FTC Requirements from one another.
Remap Nodes and Attributes	Aligns labels and values where Device and Measurement requirements do not correspond to DAU Ports
Push Attributes into Children	Pushes parent-wide attributes that matter to Port matching into ports
Pull Attributes up to Parents	This is used in Measurements, where the DataRate, SampleRate, and DataLength are buried in a structure that contains layers that would DSL ingestion
Trim Empty Branches	Removes empty branches that serve no purpose for DAU selection.
Tag Nodes that must be tracked	This is used to perform the actual remapping. Specifically, when a DAU and Ports are found that are a suitable replacement, the DSL indicates which replacement DAU and Port tags correspond to the faulty DAU and Port tags
Combine Data Using PortMappings	Utilize the PortMappings to move the data from Measurements, Devices, and DataStreams into the DAU structures

The transformation necessary to align Excitation of Ports between the DAU Ports and Device Ports in the remapping step was very complex. To that end, the following rules were developed over time discussing the problem with SwRI and the corresponding transformations are defined in the domain-specific JSON configuration:

- A DAU Port either has or does not have a corresponding excitation port for a signal conditioning port. The connection of this port is not formally indicated and the presence of it is not formally designated in MDL other than informal informative Port descriptions and names, so for the purpose of evaluation SwRI added an *ExcitationPortIsPresent* flag to DAU Ports that had a corresponding excitation port.
- A Device optionally indicates if its Ports have internal excitation or require external excitation by the DAU.
- A Device that has internal excitation can be matched with a DAU Port that does or does not provide excitation
- A Device that requires external excitation must only be matched with a DAU Port that provides excitation.
- A Device that doesn't indicate excitation can be matched with a DAU Port that does or does not provide excitation.

In this Section we will use simplified JSON (instead of XML) examples which result in a subset of data that can be provided to the Resource DSL to demonstrate the transformations instead of XML. Trivial transformations such as removing ignored or empty nodes, pulling attributes up to parents, and tagging nodes have already been performed. An example of the devices in the input FTC can be seen in Figure 36. In relation to the DAU, we simply need to know the values the replacement DAU can have that are suitable for the device.

```

1  {
2    "Devices": [
3      {
4        "name": "device0",
5        "ExcitationSource": "Internal",
6        "Ports": [
7          {
8            "name": "device0port"
9          }
10       ]
11     },
12     {
13       "name": "device1",
14       "ExcitationSource": "External",
15       "Ports": [
16         {
17           "name": "device1port"
18         }
19       ]
20     },
21     {
22       "name": "device2",
23       "Ports": [
24         {
25           "name": "device2port"
26         }
27       ]
28     }
29   ]
30 }
```

Figure 36: Raw Simplified FTC Device Excitation Definition

The transformation rules result in the following occurring, keeping things from the perspective of the DAU:

1. Since *device0* has an internal excitation source, it can be matched with any DAU port. So from the perspective of the DAU, any Port excitation value from [true, false] is suitable.
2. Since *device1* requires external excitation, it can only be matched with a DAU port that provides excitation. So from the perspective of the DAU, only a Port excitation value of true is suitable.
3. Since *device2* does not specify excitation, it can be matched with any DAU port. So from the perspective of the DAU, any Port excitation value from [true, false] is suitable.
4. Since the *Ports* are interested in excitation and not the Device itself, that excitation value must then be pushed down to each child port.

This results in the JSON file in Figure 37.

```
1  {
2    "Devices": [
3      {
4        "Ports": [
5          {
6            "name": "device0port",
7            "ExcitationSource": [true, false]
8          }
9        ]
10     },
11     {
12       "Ports": [
13         {
14           "name": "device1port",
15           "ExcitationSource": [true]
16         }
17       ]
18     },
19     {
20       "Ports": [
21         {
22           "name": "device2port",
23           "ExcitationSource": [true, false]
24         }
25       ]
26     }
27   ]
28 }
```

Figure 37: Aligned FTC Device Excitation Definition

In addition to the device data, the DAU Ports require some modification to align properly in a way that the DSL can handle them. The JSON document in Figure 38 is representative of the excitation information we receive from the DAU Inventory.

```

1  {
2      "Daus": [
3          {
4              "Ports": [
5                  {
6                      "name": "dauPort0",
7                      "ExcitationSourceIsPresent": null,
8                  },
9                  {
10                     "name": "dauPort1",
11                     "ExcitationSourceIsPresent": null,
12                 },
13                 {
14                     "name": "dauPort2"
15                 }
16             ]
17         },
18     ]
19 }

```

Figure 38: Raw Simplified DAU Port Excitation Definition

In this case, we must do the following to align for adaptation, which is encoded in the domain-specific JSON rules:

1. Since the presence of *ExcitationPortIsPresent* indicates the property and not the value, the value must be changed to *true* for *port0* and *port1*. (We could have requested SwRI update this but we wanted to limit changes on their end that we could avoid).
2. Since *port0* does not contain an *ExcitationPortIsPresent* attribute, it does not have an excitation port. To allow a proper comparison, *ExcitationPortIsPresent* must be added with the value *false* indicating it does not provide one.
3. Since *ExcitationPortIsPresent* does not match what the devices indicate, it must be renamed to *ExcitationSource* for each port.

This results in a JSON representation similar FFF which now represents the DAU Port Excitation properties aligned with the Device Port Extension properties. In this case, the DSL would be able to match it as follows:

- The *ExcitationSource* on device0port must be true or false. This can be mapped to dauPort0 as its *ExcitationSource* value is true.
- The *ExcitationSource* on device1port must be true. This can be mapped to dauPort1 as its *ExcitationSource* value is true.
- The *ExcitationSource* on device2port must be true or false. This can be mapped to dauPort2 as its *ExcitationSource* value is false.

```

1  {
2      "Daus": [
3          {
4              "Ports": [
5                  {
6                      "name": "dauPort0",
7                      "ExcitationSource": true
8                  },
9                  {
10                     "name": "dauPort1",
11                     "ExcitationSource": true
12                 },
13                 {
14                     "name": "dauPort2",
15                     "ExcitationSource": false
16                 }
17             ]
18         },
19     ]
20 }

```

Figure 39: Aligned DAU Port Excitation Definition

Let us assume Figure 36 and Figure 37 are part of the input FTC. In order to get the FTC Requirements or DAU Requirements, the operations described above would be placed in their corresponding transformation configuration, not the global one, as seen in Table 14.

Table 14: FTC and DAU requirements with different transformation configurations

Operation	FTC Requirements Operation	DAU Requirements Operation
Trim Ignored	Remove instances of <i>ExcitationSourceIsPresent</i> since it only applies to DAU Requirements	Remove instances of <i>ExcitationSource</i> since it only applies to FTC Requirements
Remap, Push, and Pull	Perform the operations that transformed Figure 36 into Figure 37	Perform the operations that transformed Figure 38 into Figure 39
Combine Data	Utilize the PortMapping to relocate values in each Device Port to the corresponding DAU Port	Not Applicable since it is self-contained.

```

1  {
2    "Daus": [
3      {
4        "Ports": [
5          {
6            "name": "dauPort0",
7            "Measurement": {
8              "SampleRate": {
9                "Min": 100,
10               "Max": 1000
11             },
12             "DataLength": {
13               "Min": 10,
14               "Max": 24
15             },
16             "DataRate": {
17               "Min": 1000,
18               "Max": 32768
19             }
20           }
21         ]
22       },
23     ],
24   ]
25 }

```

Figure 40: Transformed Flight Test Configuration Measurements

The DAU Requirements would reflect Figure 39. The FTC Requirements would be similar, however, the *ExcitationSource* values from Figure 37 would be mapped based on the PortMappings, resulting in two DAU configurations: One defined in terms of how it must be used, and another defined in terms of how it can be used. The matching is typically relatively straight-forward. However, after transformation, the measurements result in the multi-layered Port attributes for the FTC Requirements in Figure 40 and the DAU Requirements in Figure 41. The Resource DSL section goes over how these matches are performed. The

faulty DAUs in the FTC Requirements have the attribute *BBNDauFlaggedForReplacement* in their *GenericParameter* indicating they are faulty and need to be replaced, which indicates to the Resource DSL which DAUs must be replaced.

4.3.2.3.1.2 Adaptation Insertion

Although an attempt was made to isolate adaptation from the domain, it proved to be more difficult than ingestion and would have required significantly more sophisticated rules and instructions to cover all the scenarios necessary. Although we utilized the tagged data and references in transformed nodes to reference their original nodes and raw objects to determine the location of

```

1  {
2    "Daus": [
3      {
4        "Ports": [
5          {
6            "name": "dauPort0",
7            "Measurement": [
8              {
9                "SampleRate": 400,
10               "DataLength": 12
11             },
12             {
13               "SampleRate": 100,
14               "DataLength": 16
15             }
16           ]
17         ]
18       },
19     ],
20   ],
21 }
22 }

```

Figure 41: Transformed DAU Inventory Measurements

insertion, a number of hard-coded exceptions such as creating additional measurements and modifying the structure of the replacement DAUs had to be made for examples we got shortly before evaluation resulting in a domain-specific adaptation process.

4.3.2.3.1.3 Validation Checks

As stated earlier, we took an iterative approach to integrating with the MDL domain. A robust validation mechanism proved instrumental in having a successful evaluation, with much of it ensuring that we were on the same page with SwRI on a number of topics. The data was presented as a colored ASCII chart to indicate where failures occurred since examining the XML manually proved to be a time consuming task with the number of iterations that were performed for testing and execution.

We initially desired to get a clear and concise idea of what values were valid for the FTC Usage, FTC Requirements, and DAU Requirements since although the MDL specification indicated much of this, there were a number of instances where cross-dependencies between properties were not captured. In order to do this, we utilized the Drools [38] rule system to define expected values for the three different data sets in an Excel spreadsheet that could then be automatically used to validate the values, value multiplicities, and value types we were getting for the three types of configurations. This was performed in a domain-agnostic manner, utilizing the Java object that was serialized into JSON for the Resource DSL. Domain-agnostic checks were also done for dangling references where ID and IDREF values did not match up as expected.

Additional domain-specific checks were added to the FTCS to validate the integrity of the results. This involved validating the solution against the initial PortMappings, FTC Requirements, and DAU Requirements. Some of this validation was also applied to the input FTC examples SwRI provided us which helped indicate inconsistencies with the examples and where we needed further discussions to understand how we should handle scenarios.

4.3.2.3.2 Resource DSL

The Resource DSL was used to facilitate the search for replacement DAUs, and to generate a description of *how* the new DAUs replace the old DAUs by providing a port-level mapping from old to new. This application is outside the scope of what the DSL was originally defined for (modeling resource provisions and requirements of highly configurable systems), and so required a rather elaborate encoding and some surrounding infrastructure.

At a high-level, the search is conducted by mechanically translating the DAU search problem into a set of Resource DSL programs. Each generated Resource DSL program corresponds to a candidate set of replacement DAUs, and each program is executed in sequence. If a generated Resource DSL program has a viable configuration, then the corresponding DAUs can be used to satisfy the request, and the output of the DSL program can be used to generate the required port mappings.

More specifically, the input to the search is three files in JSON format:

- A *DAU Inventory* in the JSON format described in the previous section. The inventory contains all of the DAUs that can be used to satisfy a request. The DAUs in the inventory

are inherently variational since many of the ports' attributes can be configured to work in different ways or can accommodate a range of values.

- FTC Requirements, also called a *Swap Request*, that includes a set of DAUs to be replaced. The DAUs in the request are also potentially variational, but typically more constrained since they describe how the DAUs were already being used (i.e. they were configured in a particular way).
- A *Rules File* that specifies general properties that should apply to all port attributes. These rules may be equations, for example, specifying that the *DataRate* attribute is calculated as the product of the *SampleRate* and *DataLength* attributes, or they may be compatibility relationships, for example, specifying that a port whose *Direction* attribute is *Bidirectional* may also be used as either an *Input* or *Output* port.

Given these inputs, the search algorithm generates a sequence of Resource DSL programs to execute. Each DSL program corresponds to a candidate set of replacement DAUs. These candidate sets of DAUs are considered in increasing-cost order so that the first viable set we find is known to be the least expensive (cost is our primary metric for ranking replacements).

The generated DSL program is a somewhat elaborate encoding of the DAU Swap problem that allows the output of the DSL program's execution to be used to construct the required port-mapping, if the execution is successful. The DSL program is generated as follows:

1. The *request DAUs* from the Swap Request are modeled as a set of *resource requirements*. Specifically, for every port, every port *attribute* in the request DAUs is modeled as a resource that must be provided by the replacement DAUs. Port attribute specifications have four forms, which get translated into constraints in the Resource DSL as follows:
 - An attribute must have *exactly* a particular value → a constraint that simply checks whether the corresponding resource has that value *or* a value defined as compatible in the Rules File.
 - An attribute must have *one of* a finite set of values → a constraint that checks whether the corresponding resource has any of those values or any value defined as compatible with one of those values in the Rules File.
 - An attribute must be within a *range* of values (specified by a low and high value) → a constraint that checks whether the corresponding resource is in that range.
 - An attribute consists of a group of sub-attributes → recursively generate constraints for the sub-attributes.
2. The *candidate DAUs*, which are a subset of the DAU inventory, are modeled as a set of resource provisions. The encoding of resource provisions is much more elaborate than the encoding of resource requirements since (1) only some of the ports in a candidate DAU may be used (whereas all of the ports in the request DAUs must be satisfied), and (2) we need to keep track of *how* the candidate DAUs satisfy the request (that is, how to configure the DAUs, and how to assign the ports). In the Resource DSL, we model these aspects with variability in the resource model. If the DSL model is viable, then the viable configurations

returned by the DSL's semantics can be used to address (1) and (2) above and build the corresponding response. Specifically, we encode the provisions as follows:

- Port attribute specifications are encoded as described above for attributes, except that *one of* and *range* constraints are modeled by *choices*. A choice is the fundamental unit of variation in the Resource DSL and is used to model two possible values that can be selected between based on a configuration option (more precisely, a boolean formula of configuration options). Attributes with many possible values can be modeled as nested choices with different configuration options, and the values of these configuration options for a viable model indicates which alternatives can be chosen to satisfy that part of the model.
- In order to track which ports in the candidate DAUs are used, and moreover to track *which* ports in the candidates are used to satisfy *which* ports in the request, we introduce a configuration option for every pair of a candidate and request port. That is, if C is the number of ports in the candidate DAUs and R is the number of ports in the request DAUs, we will have $C \cdot R$ configuration options. If a given configuration option is turned on, then its corresponding candidate port is used to satisfy its corresponding request port.
- All *equations* specified in the Rules File are added to any port provisions with the relevant properties. Such equations may or may not be used by the requirements; if not, they will have no effect on the output of the model.

After generating a Resource DSL program for a candidate set of DAUs, as described above, the DSL program is executed. If the result of this execution is that there are no viable configurations of the model, then the next program is generated for the next candidate set of DAUs. Otherwise, if there is a valid configuration, then the values of the configuration options are used to generate the response.

The output of the search consists of two files:

- The *Swap Response* is a file in a similar JSON format as used for the DAU Inventory and Swap Request. It consists of the fully configured DAUs that satisfy the requirements of the Swap Request, extended with additional attributes describing which ports in the request DAUs are satisfied by each port in the response DAUs.
- A *Metrics File* consisting of various metrics for the search that was performed, used in our evaluation.

DAU swap requests are performed iteratively. Unused ports on replacement DAUs are added back into the inventory after each iteration, and such ports are considered “free” to use (since the corresponding DAUs are already in the system) in future iterations. We always include all currently available free ports in the model generated for any subsequent set of candidate DAUs, allowing these ports to satisfy requirements, when possible.

Since the search process described above requires the execution of a large number of highly configurable Resource DSL models to be generated and executed, this challenge problem was a major driver of *performance improvements in the DSL semantics*. Additionally, there were several

opportunities for *domain-level optimizations* that significantly reduced the size and number of DSL models that we needed to run.

The first domain-level optimization we performed was to perform some basic filtering on sets of candidate DAUs to ensure that we only generate and execute DSL models for subsets of the DAU inventory that have a chance of succeeding. Specifically, every port within a DAU has an attribute called *PortType* that describes the basic functionality provided by the port. Before generating a DSL model for a candidate DAU, we simply count the number of ports of each *PortType* in the request, and ensure that there are enough ports of that *PortType* in the candidate set. If not, we skip it and continue to the next candidate set. We call this basic optimization *candidate filtering*.

The second domain-level optimization is quite a bit more elaborate. Frequently many ports in a DAU have the same attributes. Therefore, rather than introduce resources and configuration options for individual port attributes, as described above, we instead group ports with identical attributes together into *port groups*. That is, we model port usage and attribute configuration at the level of port groups, rather than individual ports, significantly reducing the size and variability of DSL models. However, we still must allow the ports in a group to be broken up and configured differently to satisfy different parts of a request. This is enabled by repeating the provided port groups for each required port group and maintaining a global count of available ports from each group to ensure that we never use more than are available.

At the level of the Resource DSL's semantics, the major optimization for this phase was integrating our recent work on aligning incremental SMT solving with variational execution. Variational execution makes heavy use of an SMT solver under the hood to manage variation contexts and track the viability of different configurations. In previous work, and in Phase 2 of this project, the underlying SMT solver is used as a global black box. However, some modern SMT solvers, such as Microsoft's Z3 solver, expose an "incremental" interface that allows directly manipulating and querying the state of the solver.

We optimized the semantics of the Resource DSL by essentially aligning the variational execution algorithm to the interaction paradigm of the incremental solving, for example, the semantics of a choice between two alternatives becomes:

- push the choice's condition into the SMT solver's state,
- recursively execute the choice's first alternative,
- pop the solver's state,
- push the negation of the condition to the solver,
- recursively execute the choice's second alternative, and
- pop the solver's state.

By aligning the semantics in this way, we (1) take advantage of a huge amount of engineering work on optimizing incremental SMT solving, and (2) can directly reuse the internal solver state during variational execution, significantly reducing redundancy and minimizing the overhead of interacting with the solver.

Aside from this research-oriented optimization, we also significantly improved the performance of the Resource DSL semantics through standard software engineering practices, such as iteratively profiling the code and refactoring to eliminate hot spots.

4.3.2.4 Evaluation

We worked with SwRI to design an evaluation workflow that maximized the use of their examples. This resulted in every DAU created being added to the DAU Inventory and an iterative approach to DAU replacement. This iterative approach can be seen in Figure 42, where we would continue to attempt to create a new solution until the DAU Inventory was empty or there were no possible solutions left. The key driving mechanism was that DAUs that were moved from the DAU Inventory to the resultant FTC were marked as faulty so that a replacement attempt would be made each iteration.

4.3.2.4.1 Internal Evaluation

Our lack of familiarity with the domain largely led us to focus on the examples provided to us by SwRI. They ultimately produced 16 examples that utilized a single inventory and the iterative approach described above. In total, 97 successful adaptations were performed.

The average running time of the Resource DSL-powered search was 3.37 seconds per iteration on the system used for internal evaluation. However, this average was driven by a significant outlier; adaption #25 took 301 seconds. The median adaption time was 0.224 seconds, 93 of 97 adaptations took under 0.5 seconds, and the second longest running adaptation took 1.78 seconds. The very low execution times for the vast majority of searches demonstrates the viability of using variational execution and the Resource DSL encoding to support search for DAU swap solutions.

The running time of a variational execution is driven primarily by three factors: (1) the running time of the individual variants, which in the case of the Resource DSL, can be approximated by the size of the program since there are no loops or recursion; (2) the degree of variability since in principle every boolean configuration option doubles the number of variants to consider; (3) the amount of sharing among variants, which enables sharing of effort. Factors (1) and (2) are easy to measure. For the DAU Swap problem, the size of the program will be proportional to the total number of attributes in the candidate set and the request, and the degree of variability will be the total number of configuration options used in the program. Factor (3) is much more difficult to estimate, but also perhaps the most crucial factor since (3) is the key to preventing the state-space explosion implied by (2). Since sharing is difficult to measure or estimate, predicting the performance of variational programs remains very difficult and is a topic for future research [38].

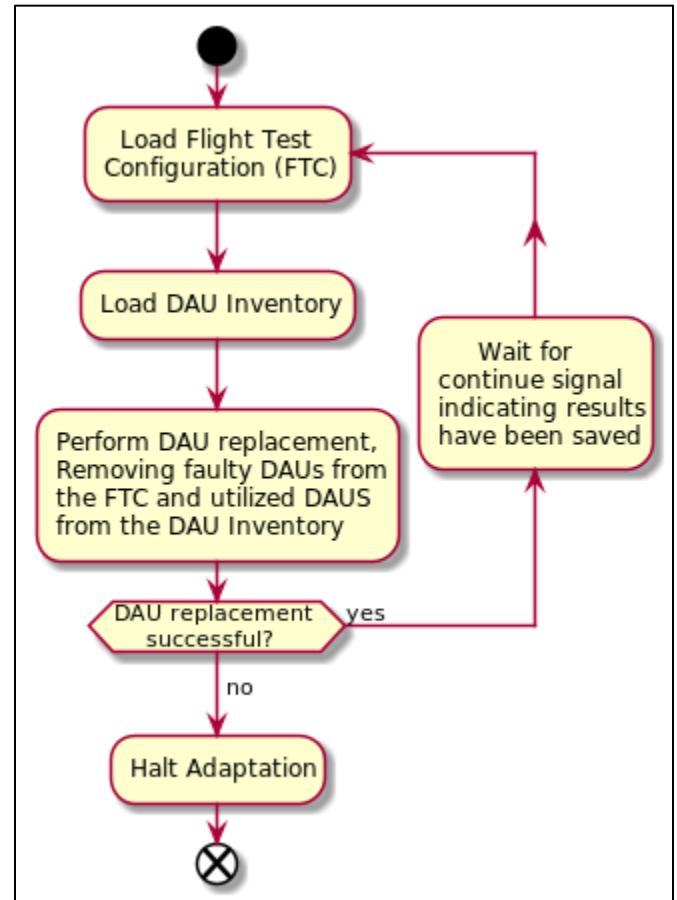


Figure 42: MDL Flight Test Plan adaptation evaluation workflow

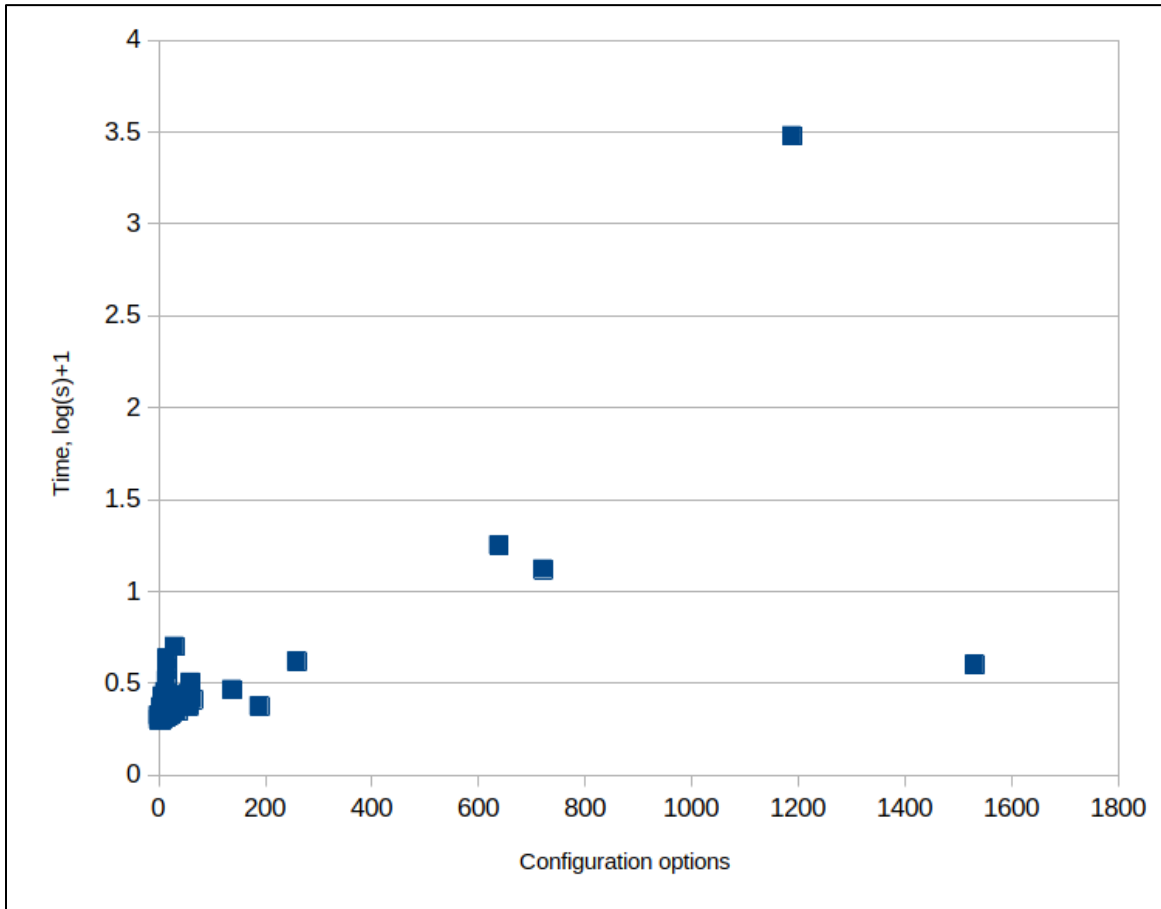


Figure 43: Execution time as a function of configuration options

In the data we collected over our 97 adaptations, the total number of attributes appears to have a minimal impact on the running time of the search ($R^2 = 2.3\%$ for an exponential regression). This is to be expected due to the absence of looping or recursion. The total number of configuration options, however, has a significant impact on running time ($R^2 = 49\%$), indicating that the size of the configuration space is a key driver in the running time, again as expected. Figure 43 shows a plot with number of configurations options on the x-axis, and search time on a logarithmic y-axis.

The impact of our domain-specific optimizations was significant. Our adaptation searches explored an average of 243 ports, but with a huge range (std. dev. 752, median 35). This was compressed into a more manageable (but still highly variable) average of 56 port groups (std. dev. 172, median 7). Additionally, we were able to ignore an average of 34 candidate sub-inventories per search (std. dev. 22, median 25) through our candidate filtering optimization.

The impact of the optimizations to the Resource DSL semantics was even more substantial. Our core optimization for this phase was tightly integrating the semantics with an incremental SMT solver. This optimization achieved $\sim 100x$ speedup in both our unit test suite and on very small

DAU swap problems (larger problems were intractable before this). Additional engineering effort to remove hotspots identified via profiling yielded an additional ~20x speedup, resulting in ~2000x speedup for the Resource DSL's semantics overall between Phase 2 (where optimization was not a priority) and Phase 3 (where it was, both according to our plan, and by necessity because of our approach to solving the challenge problem).

In addition to the SwRI-provided examples, two custom examples were created to test cost optimization and multi-DAU selection behavior. The Resource DSL only allowed selection of more than two replacement DAUs with a flag since the search space explosion resulting from it resulted in an inability to scale with the targeted inventory size with up to 100 DAUs. For this, we created two custom inventories with a limited but targeted set of DAUs. The costs were optimized so that each iteration would select a specific number of DAUs (2, 1, 4, 2) and (4, 2, 1, 2), which resulted in the expected selections and exhaustion of the DAU Inventory.

4.3.2.4.2 External Evaluation by the Voice of the Offense Team (SwRI)

The SwRI evaluation executed a single workflow with a fixed inventory of available DAUs and a faulty DAU used in the test plan which behaved as expected (see results in Table 15).

Table 15: Metrics collected from SwRI evaluation

Metric	Iteration 0	Iteration 1	Iteration 2
Total Match Dimensions	28	46	97
Candidate Sub-Inventories	121	106	79
Explored Port Groups	56	99	195
Required Port Groups	4	4	4
Ignored Sub-Inventories	0	0	0
Required DAUs	1	1	2
Explored Ports	246	397	844
Explored Sub-Inventories	10	15	22
Total Config Dimensions	26	39	81
Required Ports	28	28	28

4.4 Handling Changes in Data Representation and Schema

4.4.1 Updating Database Application to Schema Updates (Phase 2, CP1)

4.4.1.1 Motivation

Relational database schema form a significant aspect of an application's supporting ecosystem. As an ideal mechanism for enabling robust cross-application communication and providing fast access to large centralized data stores, database schema are often the focus point for change across an organization's application map. Many types of changes may be made to database schema. One common type of change is due to composition and decomposition of tables to improve performance or security. These changes often break SQL statements embedded in applications, resulting in increased manual effort to perform impact analysis, code and SQL repair and testing. Hence, automation in any aspect of this effort would significantly reduce an application's total cost of maintenance and improve its longevity.

Columns from multiple tables are often combined (or composed) into a common table to eliminate the performance overhead of evaluating SQL joins. The order in which join statements are resolved affects performance. The database query optimizer must expend CPU and memory resources to determine and execute an optimal join strategy. Even for indexed join columns, join execution strategies carry an additional overhead at least equal to the number of rows in the query's result set. Hence, data architects and DBAs often compose columns into common tables in order to eliminate the need to use SQL statements with joins. Columns from a single table may also be *decomposed*. For example, an employee's salary table may be moved to a new table with more restricted read permissions than the original table.

An automated solution to address the impact of these changes would perform the following steps: (1) Determine the code sites that are impacted by a schema change; (2) Produce a semantically equivalent SQL statement over the modified schema; (3) Automatically repair the affected code site, including ensuring parameterized statements are set in the correct order for the new SQL statement; and (4) Validate the changes are correct. The IMMORTALS approach addresses all three components using a combination of relational learning and program analysis and repair. These components are described next, beginning with the component that learns SQL statements.

4.4.1.2 CP Description

For this CP, the platform under test was the TakServer application, specifically the TakServerDataManager Java module that contains 10 DFUs that each returns CoT records matching a set of conditions. The queries

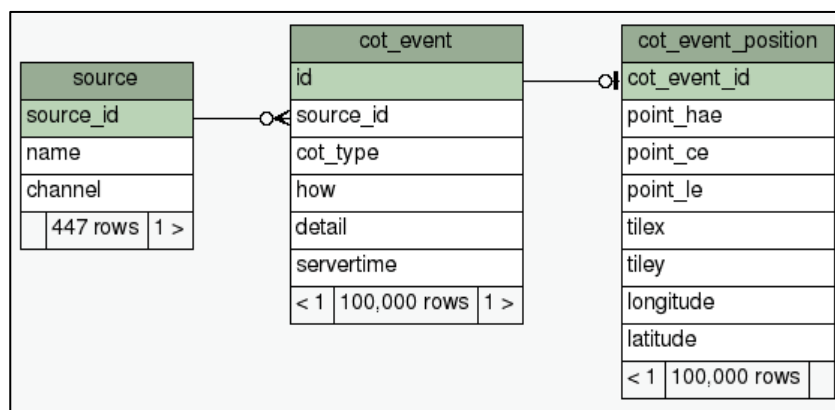


Figure 44: Baseline CoT Schema

used by the DFUs represent a set of increasingly complex queries. Parameterized queries are set to variables passed into the DFU's execute method. For evaluation and testing, the baseline CoT schema shown in Figure 44 was perturbed.

Not all schema transformations can be learned in a data driven approach—the current technology can support schema changes like de-normalization, column renaming and ordering, horizontal partitioning etc. Perturbations used for testing and evaluation were therefore limited to a set of *valid* transformations. Figure 45 shows the transformed version of the baseline schema after such a valid perturbation. The goal was to learn the query against the updated schema and modify the code to the learned query.

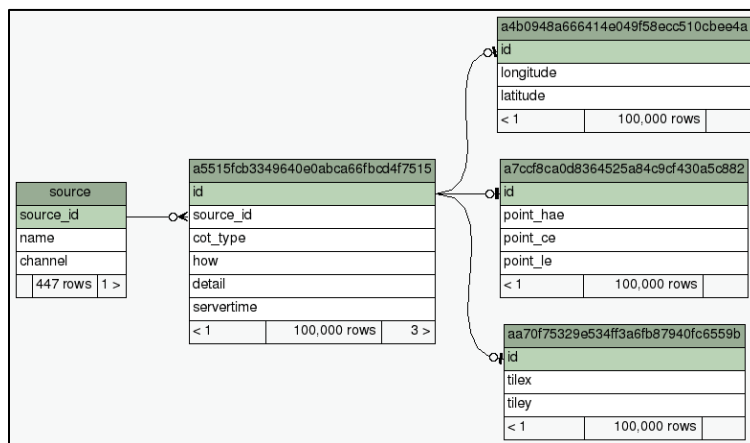


Figure 45: Perturbed Schema

The tables were populated from ADS-B data (Automatic Dependent Surveillance-Broadcast), a technology that enables aircraft to broadcast their location information to ground stations and other aircraft. Most of the columns in the figure are described in the MITRE CoT standard [13]

Because Castor cannot work with continuous data types (like float and timestamp), all data was discretized as follows:

- Latitude and longitude were discretized to tilex and tiley. These are discrete integers that represent location using the same tiling scheme as OpenStreetMap at zoom factor 16 (which provides a scaling factor useful for street-level granularity).
- Time values were chunked to 5 minute intervals in the format *yyyyMMddhhmm*; e.g., 201705071655 for May 7, 2017 4:55 pm.

In total, 100,000 CoT events over 447 discrete reporting devices (aircraft in our case) were loaded into the tables. The final results of the repair process is the repaired DFU method that contains the affected SQL statement.

4.4.1.3 Relevant IMMORTALS Mechanisms and Capabilities

Automatically deriving the query for the changed schema is key to adapting the software to work with the new database. We relied on Castor for this purpose. Castor is embedded as part an IMMORTAL AM.

4.4.1.3.1 Castor Relational Learning

The general goal of relational learning algorithms like Castor is to learn the definition of a relation given the following: A set of tuples that are members of that relation (referred to as positive examples), a set of tuples that are not members of the relation (the negative examples) and the target schema instance. Castor also requires other tuning parameters and schema metadata discussed later. The learned relation is typically provided in the form of a Datalog rule, which can in turn be converted to a SQL statement and considered as a query definition that reveals a potentially novel relationship among the entities modelled by the database. For example, given an ordered pair of person identifiers (a 2-tuple we'll call *targetRelation*), a relational learning algorithm may produce the Datalog rule shown in Figure 46.

```
targetRelation(x,y) <- yearsInProgram(x,7), publication(z,x), publication(z,y)
```

Figure 46: Sample Horn Definition Containing One Clause

In the context of an academic database, *targetRelation* may describe students in a 7 year program that have published at least one paper with their professor. This potentially describes PhD students and their advisors that may be labelled *advisedBy*(x,y). The Datalog rule is simply a Horn clause that does not contain functions and where every variable in the head of the clause appears in the body. The Horn clause above corresponds to a conjunctive query, which in turn can be viewed as the SQL statement in Figure 47.

```
select yp.x, p2.y
from yearsInProgram yp join publication p1 on yp.x = p1.x
join publication p2 on p1.z = p2.z
where yp.years = 7
```

Figure 47: Sample Conjunctive Query in SQL

4.4.1.3.1.1 Algorithm Description

Castor is an implementation of a bottom-up, example-driven relational learning algorithm, meaning it progressively generalizes from given positive examples to find a Horn definition that covers as many positive examples and as few negative examples that it can. A Horn definition is a union of Horn clauses and each Horn clause can be viewed as a conjunctive query (i.e., a simple SELECT SQL statement with only inner joins, equality checks between columns and literals in the WHERE clause, and inner joins between tables). The core learning algorithm is shown in Figure 48. The database instance I represents the current state of the database (i.e., after all (de) compositions have been applied).

The algorithm begins with an empty Horn definition (H) and a single positive example from E^+ . As it learns each Horn clause, it removes the set of positive examples covered by the clause, continuing until all the positive examples are covered by the resulting Horn definition (i.e., U is empty). The *LearnClause* algorithm consists of two parts: Bottom-clause construction and Generalization.

Algorithm 1: Castor's cover-set algorithm. Input : Database instance I, positive examples E^+, negative examples E^- Output: A Horn definition H <pre> 1 $H = \{\}$ 2 $U = E^+$ 3 while U is not empty do 4 $C = \text{LearnClause}(I, U, E^-)$ 5 if C satisfies minimum criterion then 6 $H = H \cup C$ 7 $U = U - \{c \in U \mid H \wedge I \models c\}$ 8 return H </pre>

Figure 48: Castor's cover-set algorithm

Bottom-clause construction starts by creating the head of the empty Horn definition, which consists of a relation symbol and a sequence of variables (or terms) for each column in the positive example set. This is referred to as a literal. For example, the literal for the *targetRelation* in Figure 46 may be represented as *targetRelation(x,y)*, *R(x,y)* or *advisedBy(x,y)*. Bottom-clause construction next selects a single record from the positive examples and looks for occurrences of its values through all the tables of the database. For each occurrence, the process creates a literal in the body of the Horn clause, the terms of which may be new variables, constants or references to existing variables from other literals (the last option would indicate a join across the relations denoted by the literals). The choice depends on the *mode* definition for each column. For example, the mode definition for *years_in_program* in Figure 46 indicates it should be represented as a constant. Mode definitions and predicate definitions are two additional inputs to Castor that are provided by an expert. Versions of Castor after the IMMORTALS evaluation introduced a capability to learn these inputs automatically. Predicate definitions assign high-level semantic types to columns to help determine whether it makes sense to join given pairs of columns. Note that these are not language- or database-level data types, like *int* or *long*. For example, though both *student_id* and *years_in_program* in the above example may be expressed as integers, they are of different semantic types that should not be joined in a SQL statement.

Mappings between variable identifiers and constants are maintained in a hash table as the bottom-clause construction process continues. Because each occurrence of a constant in the input database results in a new literal in the body of the learned clause, this process can generate clauses with many literals. This effect is compounded as each match generates a new set of constants that must be searched. Hence, Castor applies a maximum depth as an input parameter to limit the number of

literals. The details of how this depth metric is calculated are straightforward and described in [12]. The minimum criterion reference in Figure 48 refers to precision and recall. Values of 1.0 (perfect precision and recall) were used for both in the IMMoRTALS evaluation except for disjunctive queries where a value of 0.1 was used for recall.

The resulting Horn definition is extremely overfitted to the given positive examples and must be generalized. Castor uses a modified version of the ARMG (Asymmetric Relative Minimal Generalization) algorithm to generalize the resulting Horn definition by taking subsets from E^+ and removing literals from the body of the clause to cover E^+ . It performs this step multiple times and selects the candidate Horn clause with the best score (which is measured as the number of positive rows covered – number of negative rows by the candidate with respect to the subset of E^+). The details of ARMG and the negative example reduction algorithm are covered in [12].

4.4.1.3.1.2 Requirements and Limitations

Castor can only learn conjunctive queries or unions of conjunctive queries. While this may seem restrictive compared to the breadth of the SQL standard, conjunctive queries may represent a large fraction of queries in some types of data marts which employ a denormalized schema, like the star schema, for the purpose of supporting simpler queries such as conjunctive queries. Fortunately, data mart schema often possess two other properties that make them good candidates for relational learning: (1) The use of discrete, coarse-grained dimensions in some types of fact tables (e.g., aggregate facts that refer to discretized time intervals and geographic regions that would otherwise be stored in continuous data types; (2) A read-write preference heavily geared toward reading.

The first property is crucial to relational learning since most relational learning algorithms cannot efficiently learn over continuous data. TakServer uses an operational PostgreSQL database that defines continuous data types such as latitude, longitude and timestamps. These data types would preclude the application of relational learning. These types, however, can be discretized to support a data mart counterpart to the operational data store that can be useful for supporting fast and simplified analysis and general OLAP support. The details of this discretization are covered later in this section.

The second property is also crucial for applying example-driven relational learning, which by definition requires sets of tuples or records to drive the query learning process. DML SQL statements, like INSERT and UPDATE statements that don't produce results cannot be found by relational learning. Applications, reports and OLAP tools built around data mart schema, however, make exclusive use of SELECT statements.

4.4.1.3.1.3 Use in IMMoRTALS

IMMoRTALS focuses on vertical composition and decomposition since these transformations typically result in schema instances that are *information equivalent* with respect to the original database. This property is required in order to leverage a key benefit of the Castor relational learning tool: *schema independence*. Instances of schema R and S represent the same information or are *information equivalent* if there exists a transformation r that can map every instance I of R to instance J of S and where r^{-1} , the inverse of r , can map J back to I . r^{-1} must also be a total function (i.e., it must return a mapping for every possible input in J). Though this type of vertical

composition/decomposition is limited with respect to other types of schema changes, it represents a key change driver in databases as discussed earlier.

Schema independence is the ability to learn semantically equivalent horn definitions over schema after vertical composition and decomposition transformations are applied. This is achieved in large part by making use of the schema's inclusion dependencies (i.e., declarative foreign key relationships among tables) during the learning process, specifically within the two components of the *LearnClause* discussed earlier. This property ensures that Castor's results are invariant across (de) composition.

Note that additional restrictions existed in Castor's implementation at the time the IMMoRTALS evaluation was run, including the requirement that all database columns must be of the 'character' type and the requirement to load all schema and data to a separate VoltDB database (Castor used VoltDB to speed-up the learning process since VoltDB is an in-memory data store). Castor also prohibits the use of null values in columns. These issues may present serious challenges for future transition. The details of Castor's algorithm and the ARMG generalization algorithm that preserves schema independence are covered in the researchers' paper [12]. The next section describes how the IMMoRTALS DAS automates the tasks of application impact analysis, code/SQL repair and validation when database schema change.

4.4.1.3.2 Knowledge Builders

At application build time, the knowledge builders are invoked to perform the following functions:

1. Determine the locations of SQL statements in the application using static code analysis.
2. Isolate and execute the SQL statement to produce positive and negative examples.
3. Parse the SQL statement to learn its internal structure.
4. Learn parameter metadata for parameterized queries

Step 1 above is achieved by performing a search for invocations of the *prepareStatement* method on instances of *java.sql.Connection* using the *JavaParser* library (<https://github.com/javaparser/javaparser>). Though this is not the only coding pattern for executing SQL in Java, there are not many patterns in practice. Moreover, additional patterns can be added to the search step as needed. Once a code site that uses SQL is found, the DAS analyzes portions of the code to build the metadata shown in Figure 49. One limitation with this approach, however, is due to the limitations associated with static analysis. SQL statements that are

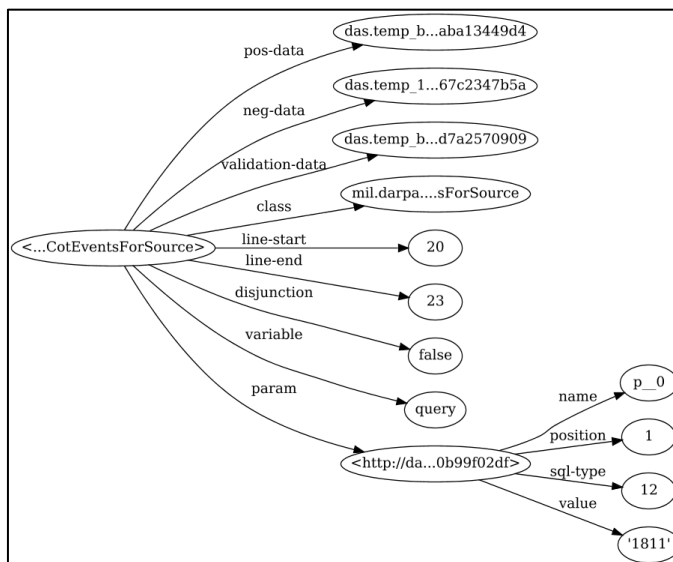


Figure 49: SQL Dependency Metadata

constructed dynamically by concatenating string literals with expressions known only at runtime can be extracted using static analysis. Most dynamic SQL statements are, however, constructed using parameterized SQL strings that are defined statically. The left-most object in Figure 49 is a data-linkage and represents a code-site where a SQL dependency exists. The *pos-data* and *neg-data* predicates indicate table names that store positive and negative examples collected by executing the embedded SQL in the baseline application. The *validation-data* predicate refers to additional data not supplied to the relational learning algorithm that can be used to validate the learned SQL. For evaluation, this dataset was equivalent to the positive examples as described later. The *class*, *line-start* and *line-end* predicates indicate the Java class name for the code site and the starting and ending source code line numbers that contain the SQL string. The remaining predicates describe aspects of the SQL statement itself and are determined using JSQLParser (<https://github.com/JSQLParser/JSqlParser>). The disjunction predicate indicates whether the SQL statements contain a WHERE clause with a disjunction. This information is provided to Castor to indicate that a union of conjunctive queries can be used to cover the positive examples. The limitations of this mechanism are discussed later in this section. The *variable* predicate indicates the Java String variable name that stores the SQL statement. Finally, for parameterized SQL statements, a *param* triple and set of other triples are created for each parameter. This captures the ordinal position of the parameters, the JDBC SQL Type constant and a legal sample value for the parameter. The value is bound to the parameterized SQL statement to produce the training data described earlier.

Learning parameter metadata so that parameterized queries can be handled in the 4th step. Example-driven relational learning does not support the notion of parameters. The general strategy for learning parameterized SQL statements is to first bind all parameters to legal values and then use the resulting data set as training data for relational learning. If Castor can learn the bound version of the query correctly, the learned query can be re-parameterized and repaired into the call site. The re-parameterization process must take into account that parameter ordinal positions may not be equivalent to the original query and that the learned query may include additional conditions in the WHERE clause that are not parameterized. This is described further below.

Test scripts written by the developer may be used to determine parameter values, but it's common to lack complete and up-to-date code coverage for all the SQL statements in an application. Since the domain of relational learning is discrete data, parameterized SQL statements can be rewritten automatically to find legal sample values for a query's parameters. For example, the query referenced in Figure 49 is shown below in Figure 50.

```
select s.source_id, ce.srvvertime, cep.tilex, cep.tiley
from source s join cot_event ce on s.source_id = ce.source_id
join cot_event_position cep on ce.id = cep.cot_event_id
where s.source_id = ?
```

Figure 50: Sample Parameterized Query

At build (analysis) time, the DAS rewrites and executes parameterized queries, such as the one shown above, to yield valid parameter values without requiring test scripts. The rewrite process executes the following generic steps on each parameterized query:

- (1) Each parameter is migrated to the end of the projection in the order it appears in the WHERE clause. The parameters are aliased as p_x where x is the ordinal position of the parameter. Note that in general the parameter may or may not already be in the original query's projection. The process described here makes no assumptions on this.
- (2) Each parameter is replaced in its original location in the WHERE clause by 'true'. In general, WHERE clauses will contain a mix of conditions that contain parameters and constants across arbitrary logical expressions. It's important to not disturb this structure or the constant values while removing the effect of the parameter so that the query can be executed. Replacing the parameter indicator ('?') with 'true' effectively removes the parameter without disturbing the original WHERE clause structure.
- (3) The resulting query is aliased as `__t1` and an outer query is written to project all the parameter aliases in order.
- (4) The DAS selects the first record from the results with the largest record count, however, results can be sampled using some other priority.

```
select p__0 from
  (select s.source_id, ce.srvtime, cep.tilex, cep.tiley,
    s.source_id as p__0
   from source s join cot_event ce ON s.source_id = ce.source_id
   join cot_event_position cep ON ce.id = cep.cot_event_id where
    true)
as __t1 group by p__0 order by count(*) desc
```

Figure 51: Rewrite of Parameterized Query

The results of applying these steps to the query in Figure 50 is shown in Figure 51.

Negative samples are produced by negating the entire WHERE clause in the bound version of the original query. At the end of the knowledge building stage, the DAS has a record of where each SQL dependency is in the application, metadata it will need later to repair the SQL statement and training data to support a relational learning approach. This process can be straightforwardly adapted to handle applications with heterogeneous relational backends by interrogating connection details within code or other markers associated with inversion of control frameworks.

4.4.1.3.3 Change Managers

When the schema is modified, the Schema Change Manager is invoked. This gives the DAS an opportunity to verify each data linkage found during the knowledge building stage to determine whether the corresponding SQL statements are impacted. For schema evolution, this process is similar to the work performed by the validation modules. In short, each data linkage is verified to ensure that its corresponding SQL statement produces the same validation dataset using the parameter bindings determined during knowledge building. If the SQL statement does not compile or returns different data, the adaptation process is triggered.

4.4.1.3.4 Adaptation Modules

The high level steps to adapting a data linkage to schema changes are as follows:

- (1) The DAS uses the available metadata and training data associated with the data linkage to invoke Castor and attempt to learn the modified SQL statement over the perturbed schema.
- (2) If the SQL statement is not parameterized and Castor learns a query with perfect precision and recall (based on Castor's return values), the DAS verifies that the learned SQL statement produces the stored validation data exactly. If so, it repairs the code site with the new SQL statement and proceeds to validation.
- (3) If the SQL statement at the data linkage is parameterized, the DAS must perform additional steps to determine the following:
 - a. Which expressions containing literals in the WHERE clause are to be bound to parameter indicators (i.e., the '?' symbol). The queries learned by Castor may include a different number of literal expressions than parameterized expressions in the original query. Moreover, the original query may have had a mix of literal and parameterized expressions.
 - b. The correct order of the parameterized expressions. In Java, parameters are referred to by their ordinal position in the SQL statement when they are set. Hence, the order of the Java set statements and parameter indicators must match. If the order does not match, the DAS modifies the ordinal numbers of the set statements appropriately.

Since Castor only supports variable character data types in database columns, the data type of the column reference in a parameterized statement cannot be used to pare down the number of possible assignments in (a) and (b) above. Moreover, since (de) compositions may alter the names of columns (and often must in order to avoid collisions), column names also cannot be used to help narrow the search space. Since the search space of (a) and (b) is constrained by the number of binary expressions in conjunctive queries, which in practice is relatively small, an exhaustive search may be performed to answer both (a) and (b). Each candidate SQL statement is executed in order to verify whether the results it returns exactly matches the validation data stored for the original data linkage. Once the correct subset of parameters and their order is determined, this information is used to potentially reorder the parameter *set* statements in Java.

The specific sequence of steps the DAS follows are:

- (1) The valid combinations for a parameterized statement are enumerated by first generating a sequence S of integer arrays where each array represents a k element subset of $\{0 \dots n-1\}$ where

k represents the number of parameters in the original SQL statement and n represents the number of binary expressions in the learned SQL statement where one side of the expression is a column reference and the other is a literal. For each integer array S_i in $S_0 \dots m-1$ where $m = \binom{n}{k}$, a bitmask of length n is created where the bits corresponding to the elements in S_i are set to 1.

- (2) A parse tree is then created for the learned SQL statement (which, recall, is fully bound without parameters). A custom expression visitor is used to traverse the parse tree and, using the bitmask created in (1), replace the appropriate expressions in the SQL statement with parameter indicators. This step is performed for each of the m bitmasks.
- (3) For each parameterized SQL statement step (2) yields, the DAS searches for the correct order of assignment with respect to the list of parameter values stored with the data linkage. Hence, the DAS creates a parameter value binding for each permutation and executes the resulting SQL statement.
- (4) If the DAS finds a candidate statement that yields the same results as the validation set, it repairs the call site with the new statement, using the discovered parameter ordering to modify the set statements.

In general, the number of SQL executions the above process generates is low as compared to the number of queries an example-driven relational learning algorithm generates.

4.4.1.4 Evaluation

Though comparing the results of SQL statements to training and validation data sets can be used to verify where a learned SQL statement is correct (or at least that it produces the same results as the original SQL statement over the unmodified database), additional validation is required to verify that the code changes are correct; i.e., that the SQL statement and JDBC parameters, if applicable, were set correctly. Hence, validation includes a set of JUnit test scripts for each data linkage. For the evaluation, a set of 10 JUnit tests were created, one for each DFU that accessed the database (see Table 16). The tests confirmed that the DFU executed and returned data.

A learning-based approach to repairing SQL carries intrinsic limitations. An important limitation is that the training process and the validation or correctness of the results depends on data comparisons. However, in general, it's possible for a learned SQL statement to compile and return the same data as collected during knowledge building and yet be semantically incorrect after a schema perturbation. The root of the issue is that in practice a data set may be covered by many Horn definitions. We observed that many learned queries returned perfect precision and recall with respect to the positive and negative examples and yet were not semantically equivalent to the original SQL statement.

Due to a technical issue in using the random function on PostgreSQL, the training and validation data sets both contained all the data returned by the SQL statement. Typically with machine learning, once the generalization error for a model and training parameters for an algorithm are quantified using the validation data set, the model is rebuilt using all of the data available. Hence, this issue did not adversely impact the ability of Castor to learn from all the data available for a SQL statement.

Though in general, preserving the split between training and validation data sets helps with automated validation techniques, in practice, this was often not found to be the case; i.e., cases where the learned query produced perfect precision and recall but was not semantically equivalent to the original query were not detected by the use of a separate validation data set when using a 70/30 split. More equal splits may produce better results, though this is not likely considering the root of the issue is that the data set supports multiple queries. Any randomized sampling technique would likely preserve this property.

Table 16: Data DFUs for Evaluation

DFU	Parameterized
CotEventsForConstantChannelJoin	No
CotEventsForConstantChannelJoin2	No
CotEventsForConstantCompoundFilter	No
CotEventsForConstantCotType	No
CotEventsForConstantMixedJoin	No
CotEventsForConstantTimeInterval	No
CotEventsForSource	Yes
CotEventsForUidAndInterval	Yes
CotEventsForXYTiles	Yes
CotEventsOnChannelInRegion	No

4.4.1.4.1 Internal Evaluation

Because there are cases where Castor may learn a query with perfect precision and recall that is not semantically equivalent to the original query, it was necessary to perform a human qualitative analysis of the results.

Table 17: Schema Evolution Trials

Perturbation/Query	Perturbation 1	Perturbation 2	Perturbation 3	Perturbation 4
CotEventsForConstantChannelJoin	Correct 5.9 Minutes	Correct 5.7 Minutes	Correct 2.6 Minutes	Correct 7.6 Minutes
CotEventsForConstantChannelJoin2	Correct 0.9 Minute	Correct 0.9 Minute	Correct 1.1 Minutes	Correct 1.0 Minute
CotEventsForConstantCompoundFilter	Incorrect 6.1 Minutes	Incorrect 6.7 Minutes	Incorrect 11.2 Minutes	Incorrect 8.0 Minutes
CotEventsForConstantCotType	Correct 3.0 Minutes	Correct 2.9 Minutes	Correct 5.0 Minutes	Correct 3.9 Minutes
CotEventsForConstantMixedJoin	Correct 7.8 Minutes	Correct 7.7 Minutes	Correct 5.6 Minutes	Correct 9.8 Minutes
CotEventsForConstantTimeInterval	Correct 3.2 Minutes	Correct 3.2 Minutes	Correct 4.3 Minutes	Correct 4.3 Minutes
CotEventsForSource	Incorrect 9.6 Minutes	Incorrect 3.1 Minutes	Incorrect 5.2 Minutes	Incorrect 4.6 Minutes
CotEventsForUidAndInterval	Incorrect 3.3 Minutes	Incorrect 3.2 Minutes	Incorrect 8.2 Minutes	Incorrect 4.2 Minutes
CotEventsForXYTiles	Incorrect* 2.6 Minutes	Incorrect* 2.7 Minutes	Incorrect* 3.0 Minutes	Incorrect* 3.5 Minutes
CotEventsOnChannelInRegion	Incorrect** 2.0 Minutes	Correct 2.0 Minutes	Correct 0.8 Minutes	Correct 2.7 Minutes

A total of 40 trials were conducted, 4 perturbations for each of the 10 queries shown in Table 16. Results are shown in Table 17. Trials marked as correct satisfied the following criteria:

1. The learned queries had a perfect F1 score with respect to the validation data.
2. The modified code compiled and passed the unit tests successfully.
3. The SQL statement and surround code passed manual inspection, which means the learned query was semantically equivalent to the original query and any parameters were set correctly in code.

Trials that passed the first two criteria but not the last were marked as Incorrect* (with an asterisk). These are interesting in that they underscore the issue with validation discussed earlier: it's very difficult to provide an out-of-band, automated mechanism for verifying the semantics of learned queries. Next to each trial outcome, the running time for the relational component of the repair process is shown. A dedicated virtual machine running Ubuntu 16.04 LTS with 10 GB of Random

Access Memory (RAM) and 4-cores was used for this experiment. The virtual machine was allocated on a macOS version 10.14.6 machine with 16 GB of RAM and a 2.9 GHz Intel i7 8-core CPU).

CotEventsForXYTiles (see Figure 52) is an example of a learned query that covers the validation data perfectly but is nonetheless semantically inequivalent with respect to the original query. We

see that the DAS correctly

determined which columns should be parameterized;

however, Castor returned two additional columns in the filter on the CHANNEL and COT_TYPE

columns. Though these don't impact the query's ability to cover the validation data perfectly, the query would

potentially fail to return additional records in the future as the data changes since it's overly constrained. Moreover, we see a strange join on POINT_CE and POINT_LE (these are the circular error and linear error for the CoT message). This is likely an error with Predicate definitions for these columns since this type of join is seen on other incorrectly learned queries. As with the overly constrained predicates in the filter, this extra join condition would also overly constrain the query and fail to return records it should in the future as new data is added to the database.

Finally, there is an interesting anomaly in the CotEventsOnChannelInRegion example. The DAS correctly handles every perturbation except the first, where it adds an unnecessary predicate to the join, again using the POINT_CE and POINT_LE columns. This predicate is absent in the other perturbations, which are semantically correct. This exception should not occur if Castor were schema invariant and, hence, should be investigated further. Other examples of this inconsistency were found in other experiments.

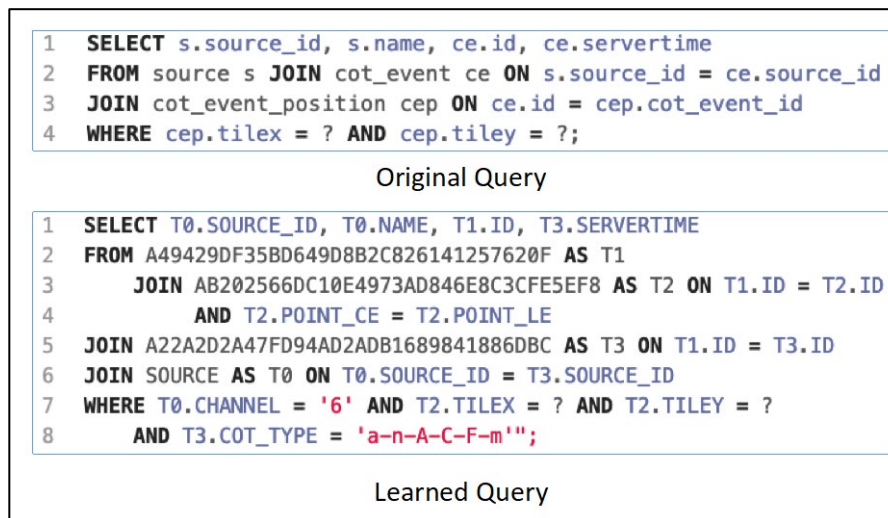


Figure 52: Example of Semantically Inequivalent Queries

4.4.1.4.1.1 External Evaluation by the Voice of the Offense Team (Lincoln Laboratory)

MIT LL was allowed to perturb the base schema shown in Figure 44. They were allowed to compose and decompose tables using the 12 different columns in the COT_EVENT and COT_EVENT_DETAIL table. Hence, the columns could be composed to one table or decomposed to a maximum of 12 tables. The total number of possible perturbations is given by the 12th Bell number as shown below:

$$B_{12} = \sum_{j=1}^{12} \frac{1}{j!} \sum_{i=0}^{j-1} (-1)^i \binom{j}{i} (j-i)^n = 4,213,597 \quad \text{Eq: 2}$$

For each perturbation, a total of 16 regression tests were executed. 10 tests were dedicated specifically to the 10 DFUs in Table 16 and the remaining tests were for the system in general. Since there exists aspects of the system that do not depend on the database, a percentage of the tests always passed even when the DAS could not repair the SQL and data DFUs after a schema perturbation. All tests completed without unexpected errors (e.g., due to virtualization, etc.). MIT executed a total of 28 tests. Figure 53 shows the results with the percentage of passed regression tests along the x-axis against the test group identifier along the y-axis. We see that almost all tests perform the same regardless of the perturbation. The only outlier is the first test group where the Adapted case failed on testImageSave, a test case unrelated to schema perturbation (since images were not being saved in the CoT schema described in this section). This failure was due to an unexpected error. The consistency of the outcome corroborates the theoretical results on the schema independency properties of Castor.

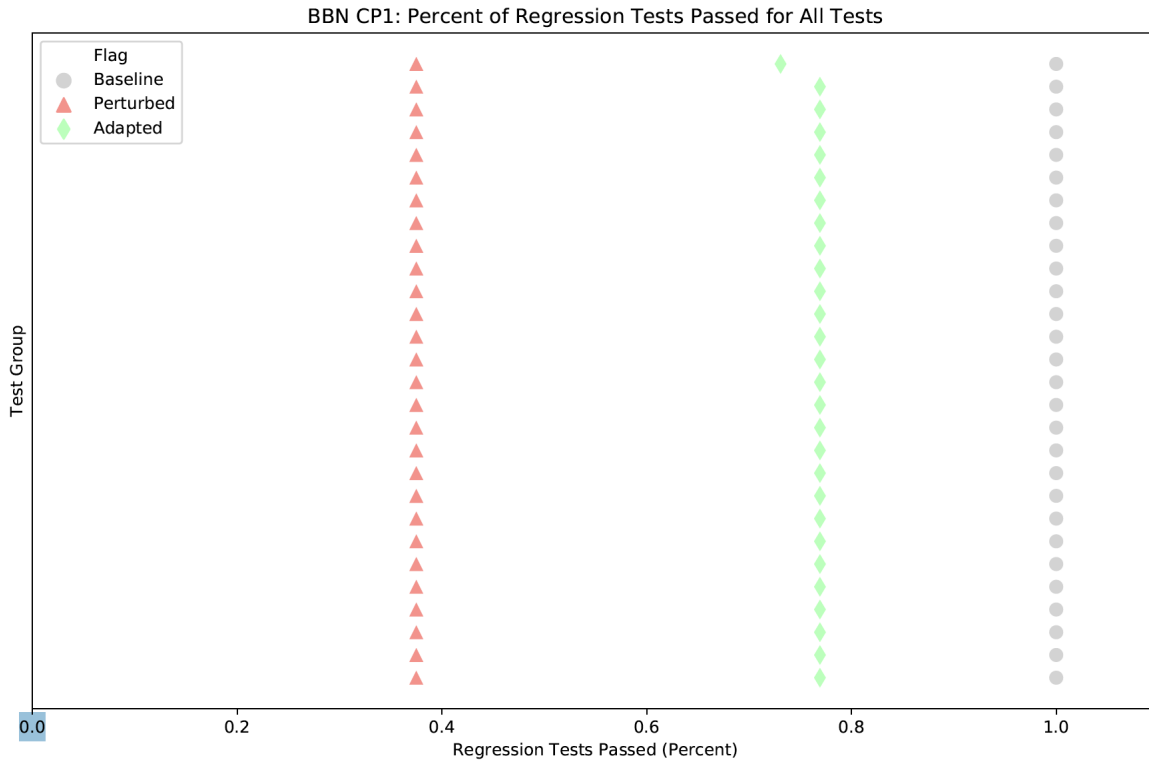


Figure 53: Schema Evolution (MIT Evaluation)

4.4.2 Updating Applications to MDL Schema Changes (Phase 3, CP2)

4.4.2.1 Motivation

A *schema* describes the structure, organization, and presentation of data. By way of example, consider the Financial products Markup Language (FpML) [51], an open-source XML Schema maintained by the International Swaps and Derivatives Association (ISDA) that defines (1) a vocabulary for financial derivatives and (2) protocols for transactions involving these derivatives. The FpML standard is used by software at many of the largest banks in the United States including JPMorgan, Goldman Sachs, Bank of America, and Citibank.

Schemas may evolve unpredictably over time to accommodate changes in the data they describe and the emergence of new use cases for that data. For example, FpML has undergone a number of significant changes that have necessitated software rewriting during its nearly 20-year lifespan (the recommended version as of February 2018 is 5.10 [52]). Schema changes can unintentionally obsolete software systems that are not manually updated accordingly. Although knowledgeable human operators may be able to define transformations sufficient to preserve otherwise obsolete functionality, these are one-off fixes made at great expense. Schema evolution therefore artificially limits the effective lifespan of software.

This CP is focused on the evolutionary pressure created by the changes in the schema, and demonstrates IMMoRTALS' ability adopt software in response to such pressure.

4.4.2.2 CP Description

The exemplar software system used in this challenge problem comprises a Java client and server that communicate MDL payloads over HTTP using SOAP¹⁰ [53] messaging protocol, as shown below in Figure 54.

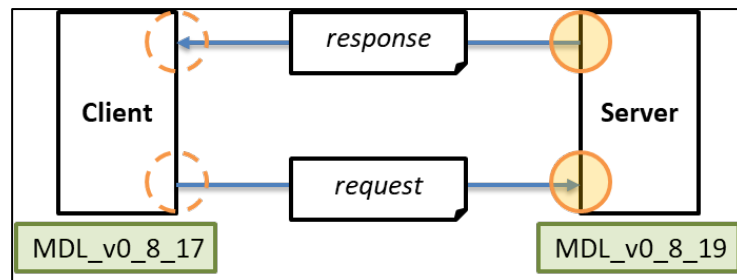


Figure 54: Exemplar client-server application used in this CP

Like FpML, MDL is an XML schema, although unlike FpML (which is a financial products vocabulary), MDL is used to describe avionics systems. In the unperturbed (and fully functional) state of the system, the client and server will use identical versions of MDL to communicate a set of input messages with one another. In the perturbed (and functionally degraded) state of the system, the client and server will communicate using different MDL schemas, which may cause issues whenever any message to be transmitted utilizes an incompatible aspect of the destination schema. In a successful CP3 run, the client system will be automatically modified by IMMoRTALS in a manner that allows the client and server to communicate using the server's preferred dialect of MDL.

¹⁰ SOAP stands for Simple Object Access Protocol

Evolutionary pressure in the challenge problem is a combination of (1) deltas between the MDL schema used by the server and the one used by the client (these schema deltas force the client to either adapt or suffer degraded performance) and (2) input documents that utilize segments of the schema having deltas. For example, if a previously optional element is made mandatory in the server's schema (but left as optional in the version used by the client), it is possible for a client to unwittingly send a malformed request to the server that will result in a runtime error. From the client's perspective, these problems typically take the form of HTTP 500 server responses, with a deeper dive revealing stack traces that indicate malformed XML exceptions thrown by the server's schema validation library.

To maximize the flexibility of the exemplar system, we use various build plugins to autogenerate client and server stubs for this system. This allows many MDL schemas (including ones not conceived of when writing the exemplar software system) to be utilized during testing without requiring client or server code to be rewritten to accommodate the customization.

4.4.2.3 Relevant IMMoRTALS Mechanisms and Capabilities

Apart from the CM that is responsible for handling the changes in the application's ecosystem that introduces the evolutionary pressure requiring adaptation (e.g., new requirements, new software that uses a new schema etc.) and the VM that is responsible for the application specific regression tests, the DAS for this CP is configured with bytecode analysis and rewriting capabilities, and a mechanism to derive a mapping between two versions of MDL schemas.

The bytecode analysis and rewriting capability leverages the IMMoRTALS ontology-based vocabulary to manage software metadata, domain knowledge, semantically annotated DFUs that can be mined for desired functionalities to be injected into the application etc.

The bytecode analysis and rewriting capability is packaged as a Gradle plugin that can be invoked with specific inputs to perform various tasks. For example `gradle bytecode` will ingest the application, perform the knowledge building analysis and emit triples representing dataflows, semantic properties attached to dataflows and designated cutpoints etc. Similarly, `gradle mine` will go through the accessible repository of annotated software components and produce triples describing their capabilities. Finally, `gradle repair` performs the proxy and API weaving needed to inject new functionality to manage semantic properties of data handled in designated dataflow. The DAS workflow orchestrates the use of the Gradle plug in through the KB and AM that perform searches (SPARQL queries on the IMMoRTALS knowledge base) and other computation in between invocation of the Gradle plugin to derive data and set up needed for the subsequent steps.

4.4.2.3.1 Knowledge Builders

The knowledge builder mechanisms supporting this CP is similar to the KBs described in Section 4.2.2.3 with the following extensions: a) we used annotations to provide additional semantic details of data handled by function that are used in our dynamic analysis to discover cut points, data flows, and data flow properties of interest, and b) we use an heuristics based technique to infer the mapping between different versions of MDL schema.

4.4.2.3.1.1 Static and Dynamic Analysis

Today, program semantics (also often described as programmer's intent) are often informally encoded informally as comments or documentation. For example, an argument to a method may appear as an array of bytes to a decompiler but the comments documenting the API might reveal that the array of bytes is in fact a UTF-8 encoded XML string that conforms to some schema. Whereas these comments and documentation are useful for humans, they aren't directly useful for automated repair tools because those tools are ill-suited to make use of unstructured content. IMMoRTALS provides mechanisms to bind explicit semantics directly to code using programmer-friendly constructs (annotations). As shown in Figure 55, the annotation `@Xml` annotation is linked to the concept of an XML schema, which is a type of Schema in an underlying ontology. The developer can define an instance of XML schema called `Mdl_0_8_17` by extending a Java class linked to the ontology. In IMMoRTALS, the semantic typing and property specification vocabulary is currently descriptive and additive/set-based in nature. There are no proscriptive (negative) assertional statements. Individual properties may be qualified by arbitrarily nested child properties (as is the case for the schema identifier in the `@Xml` construct above).

Given the mechanism of decorating code structures with programmer-friendly annotations useful for expressing application semantics, it is possible to detect the compiled in annotations in the compiled code, and use their presence to link semantic concepts to the underlying scaffold model. With additional information such as cut points identifying when data is handed over to a library calls wrapping network or storage devices, the semantic concepts linked to method arguments now opens up the possibility to determine semantic properties of data reaching these cut points (see Figure 56 later in this Section).

As in 4.2.2.3, dynamic analysis is used to discover the dataflows. For this CP, the following steps are performed as part of the dynamic analysis:

- Decompose all original method bodies discovered on the analysis classpath into CFGs (using static analysis),
- Determine a universally unique identifier for each CFG node and index the identifier in a dictionary that will be used during analysis,
- Instrument the original method bodies such that whenever a CFG node is invoked, a runtime API is alerted to the invocation and provided with a reference to the unique ID,
- Execute the instrumented application and gather execution traces from the published CFG node alerts,
- Use the traces extracted above to perform further downstream analysis of the movement of data through the application.

```
public static void processMdl(  
    @Xml(schema=Mdl_0_8_17.class) byte[] mdl  
) {  
    if(mdl[0] == 0xFF){  
        transmit(mdl);  
        return;  
    }  
    while(mdl[0] == 0xA5){  
        mdl = postProcess(mdl);  
    }  
    transmit(mdl);  
}
```

Figure 55: Annotation specifying the schema of the XML document argument

To collect the traces, a lightweight dynamic analysis runtime into the original application (the dynamic analysis runtime contains the implementation of `rt.visit` and a small number of utilities that have been carefully repackaged to prevent colliding with the classpath of applications under analysis). The implementation of `rt.visit` captures traces of program execution for each running thread within the program in a performant and thread-safe manner. The result of injecting the runtime and the new instructions into the original application is a slightly larger application that runs slightly slower than the original—the Spring-Boot app we tested in our third year of testing increased in size by 9% or 614KB and requests to instrumented service endpoints took around twice as long as those to the original uninstrumented application.

For dynamic analysis to be useful, it must be driven by test cases that exercise the code in a manner representative of realistic use cases. During the execution of these representative test cases, traces of encountered control flow blocks are dumped by the IMMoRTALS dynamic analysis runtime. An example of several lines from a trace generated during one such exemplar run for one of our challenge problems is shown below in Figure 56. The lines include method entry/exit, exception catch events and basic block entrypoint notifications. For example, the first line of the trace indicates that the `clientAction()` method in the `ClientRunner` class was invoked. The second line of the trace indicates that `block0` in that method was invoked. The third line indicates an entrypoint in the `Report()` constructor (because `block0` in `clientAction` initialized a `Report` object). Line 4 of the trace indicates a return from the `Report()` constructor. Note that accurately interpreting the block alert events as they appear in this trace requires maintaining a running copy of the stack state up to the point of occurrence.

```

1 0: enter@ com/securboration/client/ClientRunner clientAction ()V
2 1: b0
3 2: enter@ com/securboration/client/test/Report <init> ()V
4 3: b0
5 4: return@ com/securboration/client/test/Report <init> ()V
6 5: enter@ com/securboration/client/ClientRunner clientActionInternal (Lcom/securboration/clie
7 6: b0
8 7: enter@ com/securboration/client/Key getValue ()Ljava/lang/String;
9 8: b0
10 9: b6
11 10: b1
12 11: b2
13 12: return@ com/securboration/client/Key getValue ()Ljava/lang/String;
14 13: b1
15 14: b27
16 15: enter@ com/securboration/client/MessageListenerClient ping ()Lcom/securboration/mls/wsd/
17 16: b0
18 17: enter@ com/securboration/mls/wsd/PingRequest <init> ()V
19 18: b0
20 19: return@ com/securboration/mls/wsd/PingRequest <init> ()V
21 20: enter@ com/securboration/mls/wsd/PingRequest setTimestamp (J)V
22 21: b0
23 22: return@ com/securboration/mls/wsd/PingRequest setTimestamp (J)V

```

Figure 56: Trace of early stages of a SOAP/XML interaction with a remote endpoint

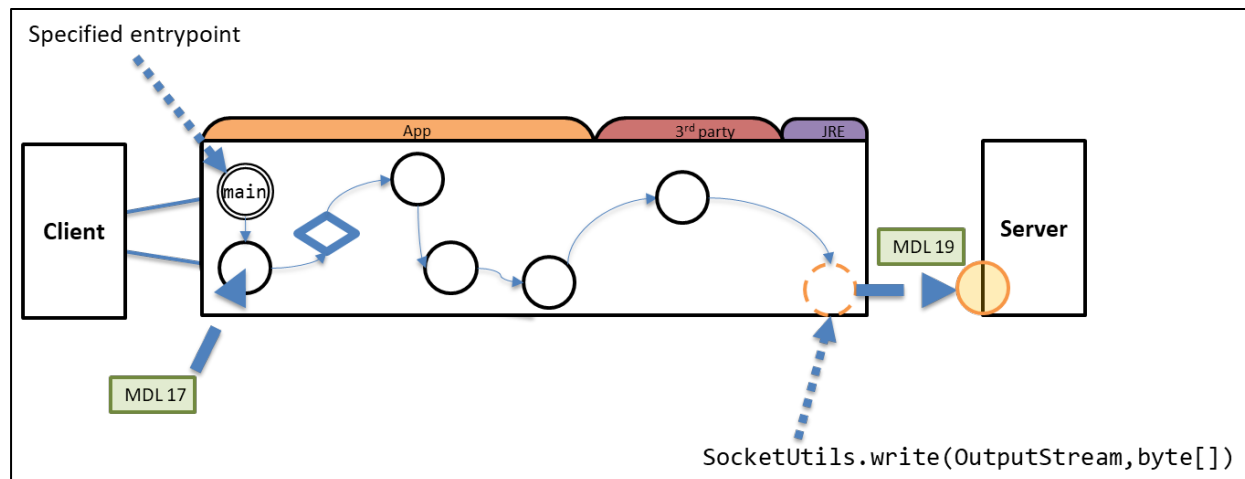


Figure 57: Hypothetical trace linking an application entry point to a network I/O cutpoint

```

387 2 invocations of java/io/OutputStream write ([BII)V
388 121 occurrences of
389 0 METHOD_BEGIN com/securboration/client/ClientRunner clientAction ()V
390 1 METHOD_BEGIN com/securboration/client/ClientRunner clientActionInternal (Lcom/securbor
391 2 METHOD_BEGIN com/securboration/client/MessageListenerClient ping ()Lcom/securboration/
392 3 METHOD_BEGIN com/securboration/client/MessageListenerClient soapRequest (Ljava/lang/St
393 4 METHOD_BEGIN com/securboration/client/MessageListenerClient httpRequest (Ljava/lang/St
394 5 METHOD_BEGIN com/securboration/client/MessageListenerClient copy (Ljava/io/InputStream
395 6 METHOD_BEGIN java/io/OutputStream write ([BII)V
396
397 1218 occurrences of
398 0 METHOD_BEGIN com/securboration/client/ClientRunner clientAction ()V
399 1 METHOD_BEGIN com/securboration/client/ClientRunner clientActionInternal (Lcom/securbor
400 2 METHOD_BEGIN com/securboration/client/ClientRunner clientAction (Ljava/lang/String;)V
401 3 METHOD_BEGIN com/securboration/client/MessageListenerClient ingestMessage (Lorg/inetpr
402 4 METHOD_BEGIN com/securboration/client/MessageListenerClient soapRequest (Ljava/lang/St
403 5 METHOD_BEGIN com/securboration/client/MessageListenerClient httpRequest (Ljava/lang/St
404 6 METHOD_BEGIN com/securboration/client/MessageListenerClient copy (Ljava/io/InputStream
405 7 METHOD_BEGIN java/io/OutputStream write ([BII)V

```

Figure 58: Actual traces connecting a clientAction() entrypoint to OutputStream.write()

Traces like the one shown in Figure 56 are valuable because they reveal actual traversals of the CFGs constructed through static analysis without falling prey to spurious or missing CFG edges. Given a set of entrypoint methods (for example, a user might define main as an entrypoint), our analysis attempts to walk the edges identified as viable in the trace to determine interactions with potentially problematic APIs (for example, interactions with the File or Socket classes) that transcend the ecosystem boundary. When the analysis identifies such a trace, it searches for problematic sequences of property assertions along the chain. An example is shown in Figure 57, which illustrates the discovery of a trace from a main entrypoint to an API that transmits data across the network to a remote server. In this example, a potentially problematic assertion chain

exists because data passed along a viable CFG traversal discovered between `main` and `SocketUtils.write` is asserted to be compliant with the “MDL 17” XML schema but is transmitted to a server that expects to receive “MDL 19” XML schema compliant documents. Note that the property misalignment detected here may not be a problem in reality (because the MDL 19 schema might be fully backward compatible with MDL 17)—determining whether it is or is not requires a deeper analysis of the schema. Note that chain discovery is essentially a graph traversal problem, and that property verification must be performed along all discovered chains (only one such chain is shown in Figure 56, whereas Figure 58 shows two actual traces chaining `clientAction()` entrypoint to the `OutputStream.write()` cutpoint).

4.4.2.3.1.2 Schema Mapping

This CP attempts to adapt a client that uses the schema version n (e.g.m MDL v17) such that objects sent by the client are accepted by a server that is coded to operate with schema version m (e.g., MDL v19). One approach is to translate version n objects to version m objects before sending. However, to do that translation a mapping between versions n and m is needed, but for this CP we assume that no mappings from n to m nor their intermediates exist. As a result, the solution approach also involved generating a mapping directly between version n and version m . The inferred mapping and translation is treated as additional metadata about the software that is used by the adaptation mechanism.

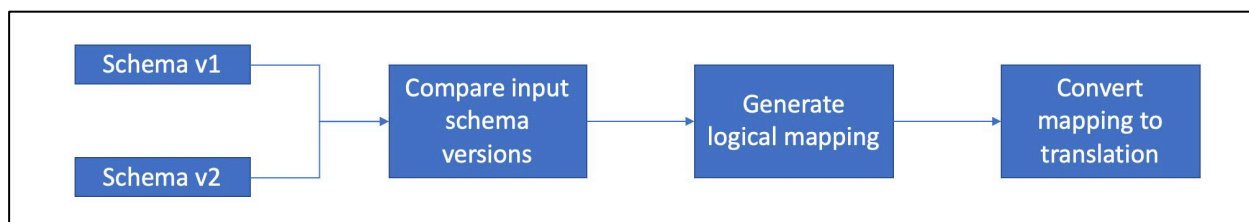


Figure 59: Schema mapping and translation process

Logically, this process involved three steps (see Figure 59)- comparing version n to version m ; generating a logical mapping between version n and version m ; and generating a translation between version n and version m . In general, discovering this schema mapping is a specialization of subgraph isomorphism, which is NP-complete. In practice, schemas are often quite tree-like and therefore mappings are often discoverable using polynomial time methods. The schema translation service accepts as inputs (1) a src schema and (2) a dst schema. It emits as output an eXtensible Stylesheet Language (XSLT) that maps documents compliant to src into documents that are compliant with dst. It also emits as output various metrics indicating the degree of fidelity associated with the mapping (for example, some schema changes, such as the addition of an entirely new element, cannot be handled by automated mapping—whereas others, such as simple element renames, are quite easy to solve in an automated manner). The following sections describe our heuristics-based schema mapping and translation technique.

4.4.2.3.1.2.1 Comparing Versions

The comparison between version n and version m is a critical first step in the mapping and translation process as it establishes the set of elements that require mapping between versions and eliminates the elements that are unchanged between the versions.

At a base level, equality between trees is established based on equality of individual elements. Put differently, a tree T_n is equal to another tree T_m if for $i = 0, 1, 2, 3 \dots$:

$$T_{n_i} == T_{m_i} \quad \text{Eq: 3}$$

While comparing two schemas does require visiting each node in the document (DOM), this initial calculation is $O(n)$ and conveys the added benefit of allowing us to index each element. As elements are visited, they can be added to an overall element list for each document. The actual calculation of identity is performed utilizing Python hash functions. Two hashes are calculated for each element (or node) and then combined to form the hash of the element itself.

First the `self_hash` is a shallow hash based on the element's name and the element's type name:

```
self_hash = hash("[node name]-[node type name]")
```

Then, the `node_type_hash` is calculated based on the node type name and the `node_hash` for each of the type's children.

The `node_hash` is calculated for each element based on the combination of the `self_hash` and the `node_type_hash`:

```
node_hash = hash(self_hash, "-", node_type_hash)
```

Once the element hashes have been calculated and indexed, document comparison itself is as simple as subtracting one document's elements from the other using Python's array subtraction functions:

```
removals = first_set - second_set  
additions = second_set - first_set
```

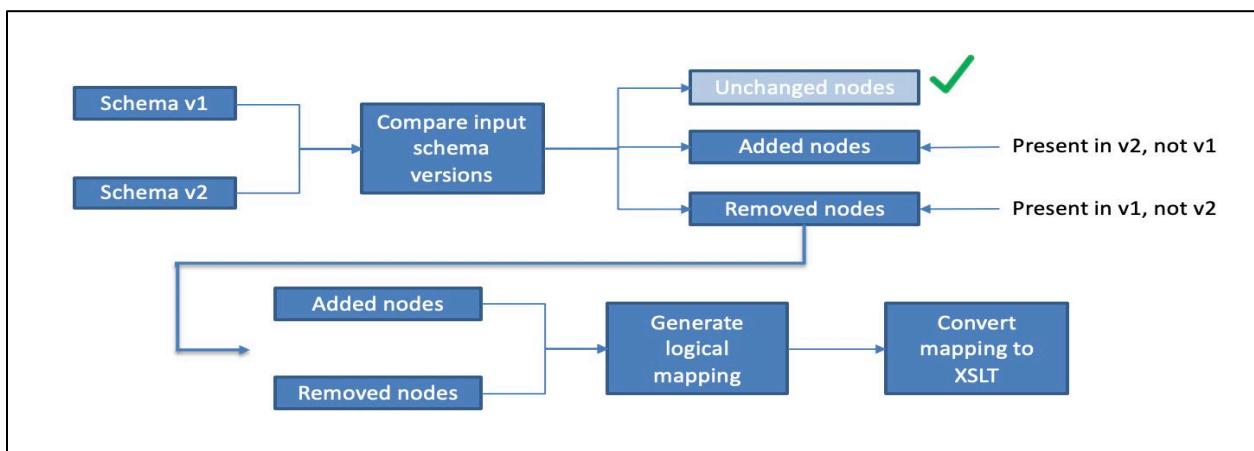


Figure 60: Output of the comparison step

The result of this initial subtraction yields two new arrays: removals and additions. Removals are elements that were present in version n but did not have an exact match in version m . Additions are the opposite. The additions and removals sets are the inputs to the logical mapping part of the process as shown in Figure 60. Elements not contained within these sets are identical between versions and can be ignored.

4.4.2.3.1.2.2 Generating a Logical Mapping

Irrespective of the translation to be performed, the logical mapping defines the relationships between versions n and m in a generic form. In simplistic terms, the logical mapping is simply a set of tuples containing references to the source (n) and destination (m) elements. Determining which elements to pair, however, requires (in the worst case) multiple passes to locate the corresponding destination element for a given source element. The first phase of the process is illustrated on Figure 61. The ultimate goal of the mapping process is to generate four classes of operations:

1. Added elements: new elements that were not present in version n . If values are required for these elements, logical defaults will be generated.
2. Removed elements: elements that do not appear in version m . Ideally, very few would exist as these represent a loss of information. It's obviously possible that data could become obsolete, but a high number of removals likely indicate a failure to properly identify a mapping.
3. Renamed elements: elements that are physically in the same place, but with a different name. For example, a `NetworkInterface` element changed to a `NetworkConnection` element.
4. Relocated elements: elements that have been moved up or down within a tree structure that is common to both versions. For example, a `NetworkInterface.Phy.Mac` element could be moved to `NetworkInterface.Mac`. This particular example would be an instance of an element moved up one level in the tree. For the purposes of preventing false positives, it was assumed that relocated elements must move up or down within the same ancestral tree and it was not possible to move elements laterally. Put differently, the source and destination must share common ancestors.

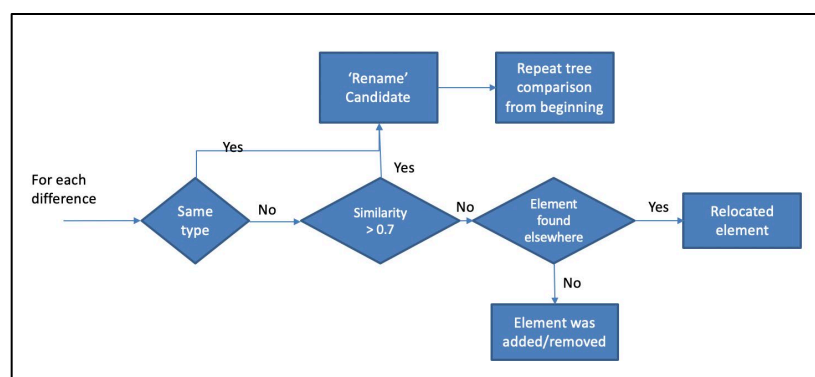


Figure 61: The mapping process

It is important to note that renamed *types* are members of a fifth operation class that is detected but has no impact on the final output¹¹. This is because type names are only a construct of the schema documents themselves. Practically speaking, given two types with the same properties but different names, an instance of one is inherently compatible with an instance of the other.

In generating the mapping, we first attempt to identify the least expensive changes first to reduce the search space before searching the entire document tree for unmapped elements.

As outlined in Figure 61, each added element is compared against each removed element, looking for similarity between the two. If they are the same type, they are considered candidates for renaming. If they are not, a similarity score between the elements is calculated as follows:

```
similarity=1.2*annotation_ratio+1*child_name_ratio+1.3*name_ratio    Eq: 4
```

Where `annotation_ratio`, `child_name_ratio`, and `name_ratio` are calculated using Python's `SequenceMatcher` comparing the type's documentation, the names of the type's children, and the type's name respectively.

If the similarity score is above a predetermined threshold (currently configured at 70%), they are considered candidates for renaming- otherwise the destination tree will be searched for an element similar to the source.

4.4.2.3.1.2.3 Generating the Translation

This is achieved trivially by converting the mapping produced by the previous step to XSLT. The adaptation mechanism can then use the XSLT to translate version *n* objects to version *m* objects.

4.4.2.3.2 Repair and Adaptation Mechanisms

Recall that the KBs included in the DAS configuration for this CP can (a) construct a control flow graph of the software system to be repaired, (b) construct a dataflow graph of the software system by traversing edges of the control flow graph from (a) determined through static and dynamic analysis to be viable, (c) identify dataflow edges from (b) by which the software system consumes or emits data assumed to be conformant to some specific version of a schema.

When the new version of the server is provided to the DAS, KBs therefore produce enough information such that the CM and VM can identify mismatches between the expected and actual schema version along the schema-dependent edges.

The adaptation mechanism packaged in the AM for the DAS configured for this CP then

- 1) identifies code units (DFUs) capable of addressing the schema mismatch identified above (XSLT transformer libraries such as Saxon)
- 2) identifies that a selected XML translation DFU requires an Extensible Stylesheet Language Template (XSLT) input that translates document instances from `vExpected` of the schema into `vNeeded` of the schema

¹¹ This is a function of XML's type system. Rather than a strongly typed system like Java, XML more closely resembles duck typing. In other output formats, this may indeed have an effect.

- 3) requests an appropriate XSLT be synthesized from the schema mapping KB described in Section 4.4.2.3.1.2
- 4) injects properly configured XSLT transformer DFU(s) into the application such that all problematic dataflow edges from (4) are handled

After code modification, the DAS tests the application for consistency and correctness (using automated unit/validation/integration tests)

For legacy application repair, we wanted to explore an approach that—rather than modifying the source that the user must compile and package—would modify and augment the bytecode directly and package the new dependencies into a new Jar file conforming to the constraints specified. The state of the application post-repair will be the same, only less manual intervention by the user. This *programmer out of the loop* poses less opportunity to baffle the IMMoRTALS user who is not familiar with the application, but at the same time this also means the augmentations are obfuscated from the user, which may or may not be an issue depending on the circumstances. This does not mean, however, that the user cannot find out what repairs occurred; the direct bytecode approach uses the same housekeeping data structure as the source-level repairs in the IMMoRTALS knowledge repository. The captured knowledge can apprise the user not only of what code structures were injected but where in the original application were they injected.

As described earlier, the basic repair activity, performed at the bytecode level for this CP, involves injecting a properly configured XSLT transformer DFUs on the client and server side code so that all data flows with mismatched schema properties are reconciled. This is insertion is done by the proxy and API weaving technique described in Section 4.2.2.3. The fact that an XSLT translator object will be able to address the dataflow property mismatch was determined by mining the annotated DFU repository and domain knowledge available in the IMMoRTALS knowledge base. The XSLT to configure the translator object was derived by the schema mapping component described in Section 4.4.3.2.1.2.

4.4.2.4 Evaluation

4.4.2.4.1 Internal Evaluation

Internal evaluation primarily focused on the schema mapping mechanism, and testing the end to end code repair capability with a provided XSLT applicable to the MDL schemas used in the client and server. This separation of concern helped parallelize the development and focus on testing the critical functionalities independently.

Figure 62 below conveys the scope of the modification to the application for an MDL v0_8_17 to MDL v0_8_19 scenario. At left in the figure is a sunburst decomposition of the classpath elements color coded by whether they were left alone (blue), modified (red) or added (green, barely visible) to the classpath. Note that the portion of the application that was modified or added is a very small percentage of the original application, which reflects the precision of the repair. At right in the figure is a listing of all deltas in the original application with red being changes to classes and green being changes to other classpath elements (in this case, both green bars correspond to XSLTs added to the classpath).

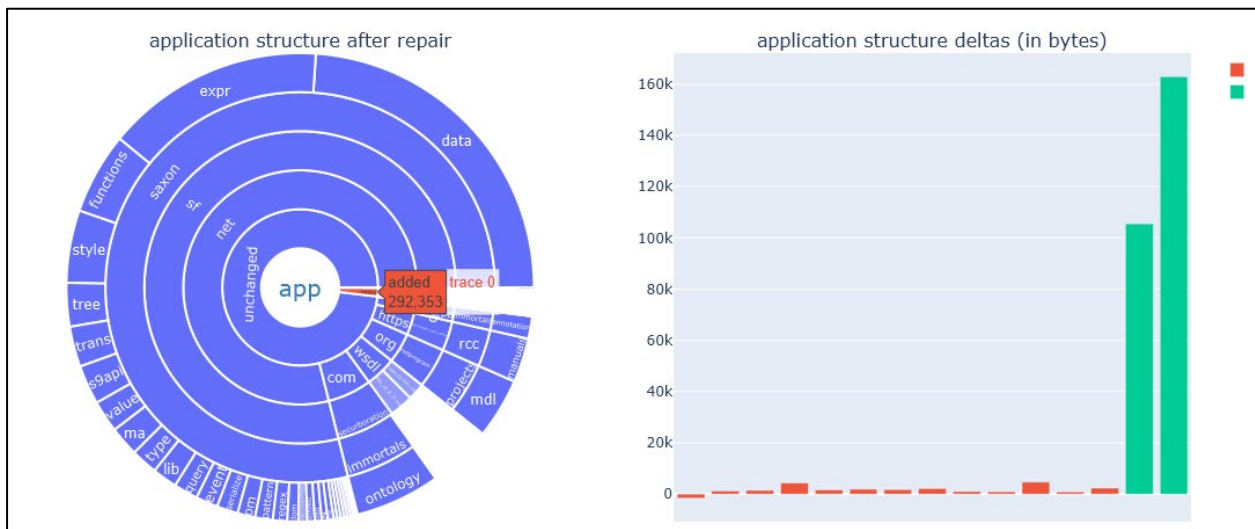


Figure 62: Bytecode diff relative to modified classpath element (L) and modified element (R)

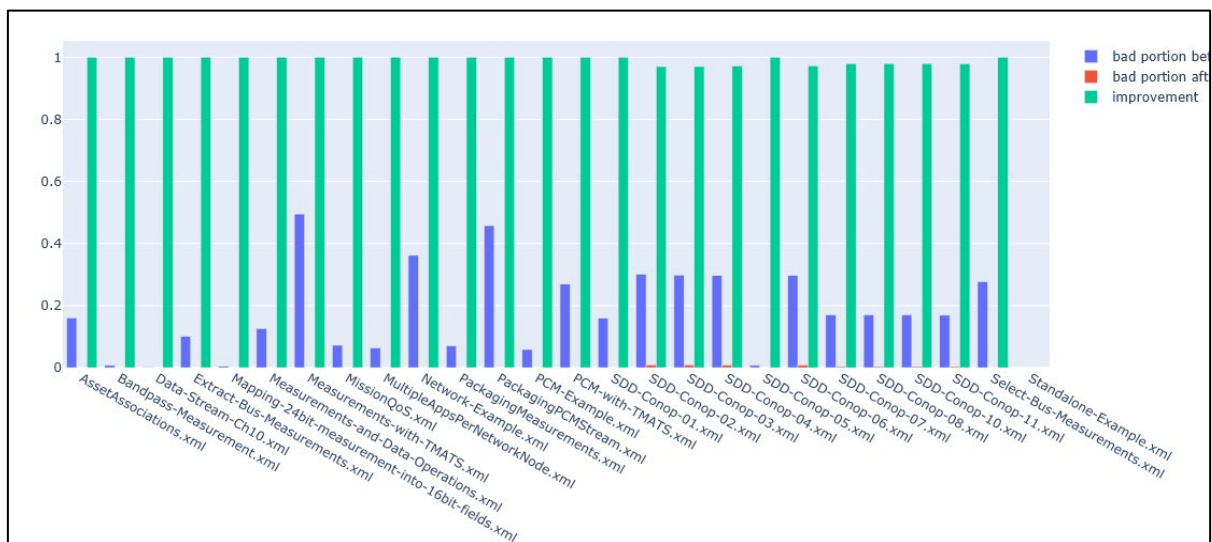


Figure 63: Idealized improvement in performance after adaptation (human-in-the-loop)

Metrics collected from the schema mapping evaluation is described below. Figure 63 and Figure 64 show the performance before and after adaptation for each of 26 MDL v0_8_17 documents in the 17-to-19 scenario's dataset. In both figures, blue bars represent the fraction of the document with XML validation errors before adaptation, red bars represent the bad fraction after adaptation, and green bars represent what was improved by adaptation (and would ideally be 1 in all cases).

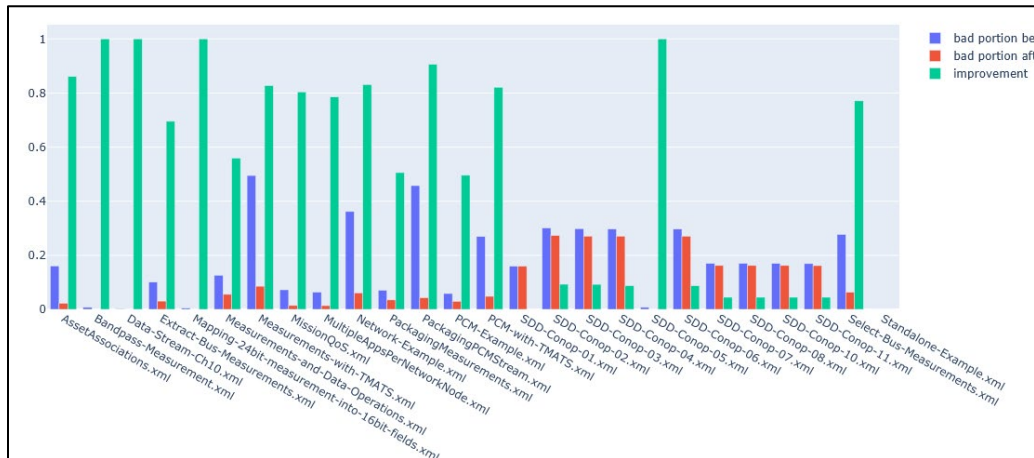


Figure 64: Improvement in performance using fully automated schema translation

The results shown above are taken from a specific MDL translation problem instance (MDL_v0_8_17 to MDL_v0_8_19). In Figure 65, Figure 66, and Figure 67 below we explore adaptation performance across a spectrum of MDL translation problems created from picking all possible combinations of two elements from a bag containing the following 11 schemas (and their corresponding document sets):

- MDL_v0_8_7 (11 docs)
- MDL_v0_8_8 (11 docs)
- MDL_v0_8_9 (11 docs)
- MDL_v0_8_10 (23 docs)
- MDL_v0_8_11 (23 docs)
- MDL_v0_8_12 (24 docs)
- MDL_v0_8_13 (24 docs)
- MDL_v0_8_14 (24 docs)
- MDL_v0_8_16 (25 docs)
- MDL_v0_8_17 (27 docs)
- MDL_v0_8_19 (27 docs)

Note that there are 121 (11^2) distinct translation problem instances within this space (the space is vastly larger if subsets of documents are selected for transmission, but to reduce the size of the space we simply select all client schema-compliant documents for transmission). 11 of these translation problems are trivial identity translations, which we include for the purposes of experimental control (the identity translation case should start with no issues and end with no issues after adaptation).

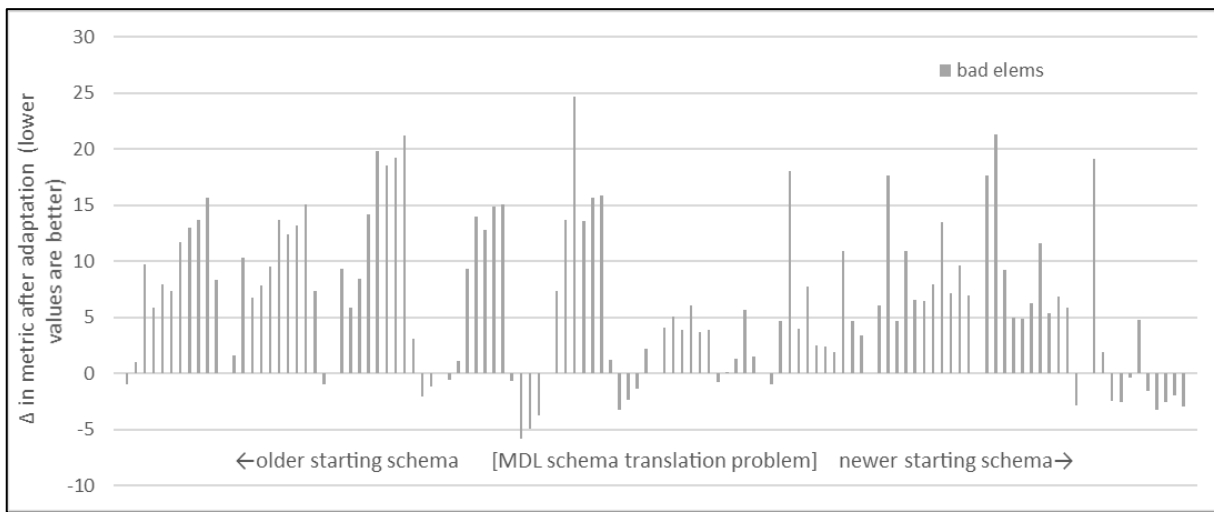


Figure 65: Change in number of bad elements across MDL translation adaptation space

Figure 65 above shows the change in the raw number of bad elements (averaged over the documents within each translation problem instance) over the 121 translation problems. A “bad element” is an XML element in which one or more child elements contain content that fails schema validation. In most cases, adaptation produces more bad elements after adaptation (positive values). The translation problem space is represented along the x axis, ordered by the starting schema version. The metrics are grouped into 11 clusters, where each cluster corresponds to the starting schema version (the leftmost cluster in the figure corresponds to MDL_v0_8_7 and the rightmost corresponds to MDL_v0_8_19). The clustering of metrics is more readily seen in Figure 66 and Figure 67 below.

Figure 66 shows the change in the number of bad lines (averaged over the documents within each translation problem instance) over the 121 translation problems. A “bad line” is a line in a whitespace-normalized version of the XML document that belongs to any “bad element” (as defined above). Here we see interesting behavior in that for older initial schemas (i.e., the destination schema is newer than the client), the number of bad lines increases whereas for newer initial schemas (i.e., the destination schema is older than the client), the number of bad lines generally decreases. In other words, the automated translation is better at converting newer documents to work with an older schema than it is at converting older documents to work with a newer schema. This is somewhat intuitive for schema changes that are additive in nature (because it is easier to deal with too much information than it is to deal with too little).

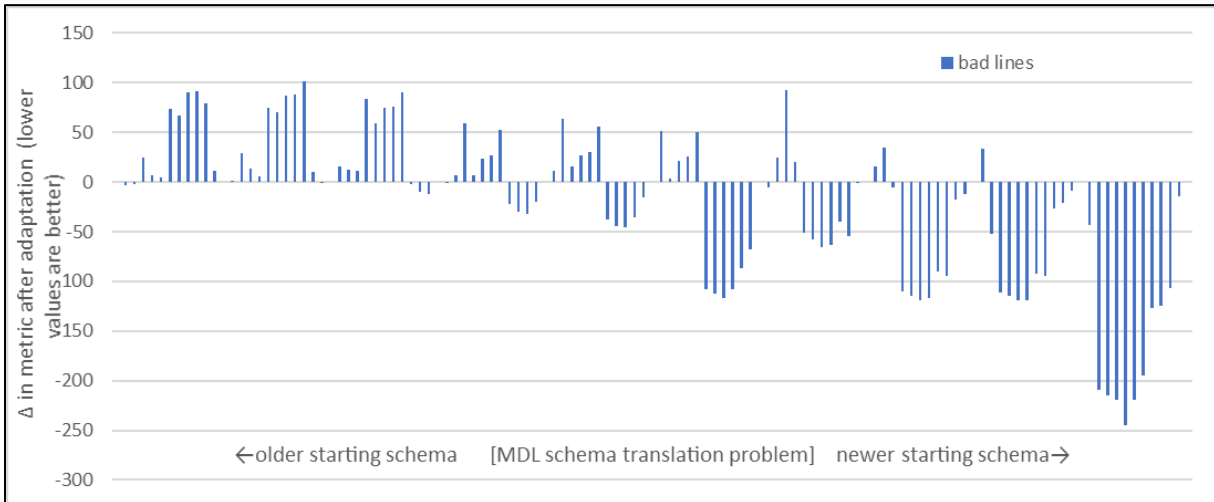


Figure 66: Change in number of bad lines across MDL translation space

Figure 67 shows similar data to Figure 66, except for the bad document fraction metric. The “bad document fraction” of an XML document is computed as the number of bad lines (as defined above) in a whitespace normalized version of the document divided by the total number of lines in that document. The same trend observed in Figure 66 is borne out in Figure 67—and for much the same reason (it is easier to remove data from a document to comply with an older schema than it is to add new data to comply with a newer one).

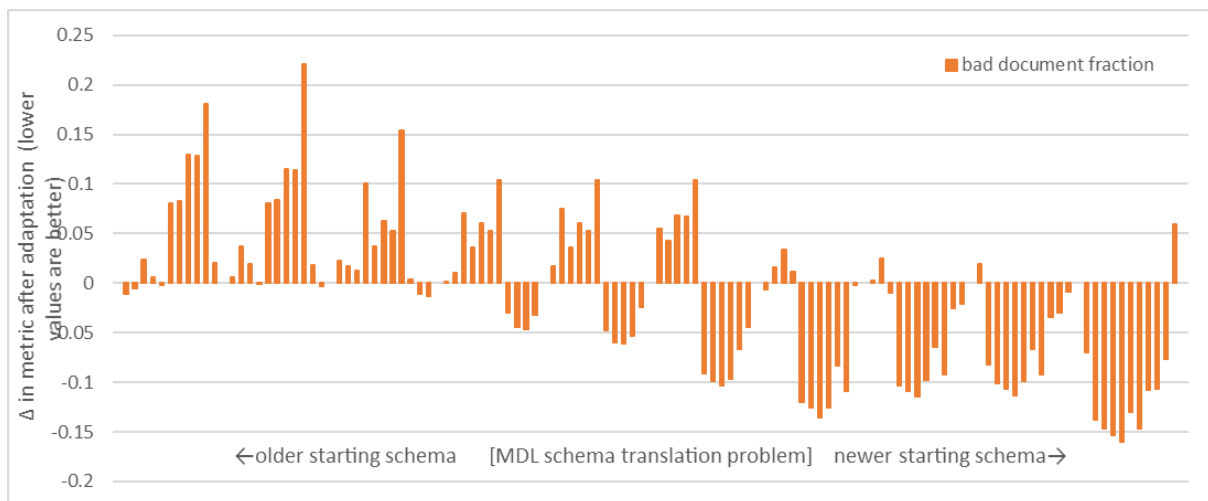


Figure 67: Change in fraction of problematic documents across MDL translation space

4.4.2.4.2 External Evaluation by the Voice of the Offense Team (SwRI)

Table 18 shows the results collected by SwRI. The bad document fraction and bad lines per document went down as expected, and the bad element count increased. Many of the specific errors could have been corrected with additional domain knowledge and a facility to import and utilize it. However, due to limited time and flexibility, we constructed the adaptation mechanisms to be domain agnostic and to apply throughout the sample of MDL versions.

Table 18: SwRI evaluation results

Metric	Adaptation Result
server Ping (millis)	Decreased by 30.23%
# tests performed/passed/failed	Unchanged
mean bad document fraction (lower is better)	Decreased by 53.05%
mean bad lines per document (lower is better)	Decreased by 55.09%
mean bad elements per document (lower is better)	Increased by 57.14%
overall elapsed time (millis)	Increased by 42.08%

We attempted to account for the most significant expected errors throughout the sample set which helped improve our number of bad elements, however, due to a focus on all examples instead of this specific one, we did not cover all of them in this scenario. Examining this specific scenario further, we discovered most of the errors fell into a few different categories. This example performed a significant amount of refactoring that moved nested elements to separate objects that each had their own ID that was referenced by an IDREF. Since the values for the IDs and IDREFs were not present in the original document, a number of errors related to it were detected. These errors can be seen in Table 19. The raw error count validating the v17 document set against the v19 XML Schema Definition (XSD) were 114 while the raw error count validating the translated document against the v19 XSD were 641. The above errors accounted for 592 of these errors, which were not present in the original documents since the relocated elements were simply unexpected or missing elements instead of a structure with a variety of potential issues. This left 49 actual errors, which is a 57.02% decrease in errors.

Table 19: Expected XML validation errors

Reported Error	Details
There is no ID/IDREF binding	There is no valid ID to reference in the IDREF
Attribute 'ID' must appear	ID missing from original document, resulting in a missing ID
Attribute 'IDREF' must appear	IDREF missing from original document
Key '*' with value '*' not found	The original document was missing a reference to a key with a corresponding element value
The value " of attribute 'IDREF' on element '*' is not valid	The " placeholder inserted to try and avoid IDREF validation errors is an invalid value
" is not a valid value for NCName	An invalid IDREF value has been used
"*" is not facet-valid with respect to enumeration	The enum used in the original document is no longer valid

The code modified is similar to that which was encountered during internal testing, and can be seen in Figure 68.

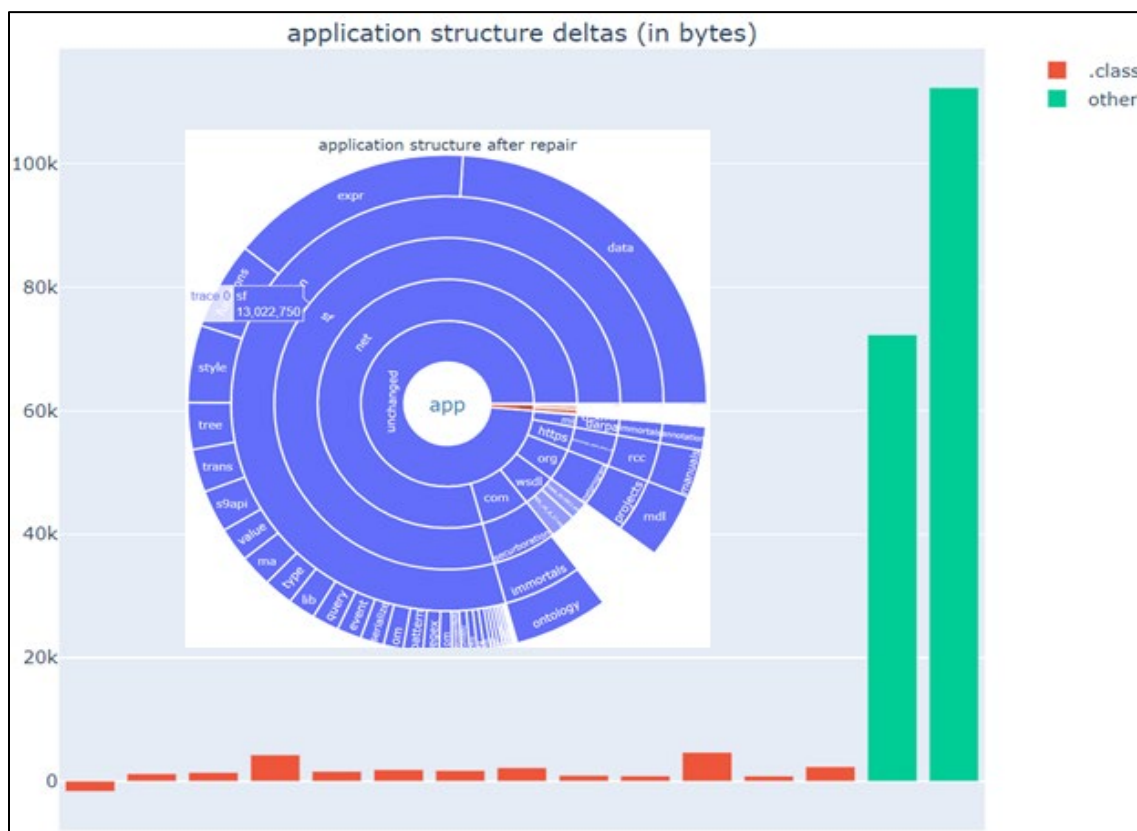


Figure 68: Bytecode modification statistics in SwRI evaluation

4.5 Handling Changes in Software Stack

4.5.1 Adapting to Changes in 3rd-Party Libraries (Phase 2, CP3)

Modern software stack is complex and tall. Even a simple application depends on multiple operating system, programming language- and/or application specific libraries. While it simplifies the development task by encapsulating frequently used tasks and programming constructs and making them readily available, it also greatly complicates the long term maintenance and upgrading of software since the libraries and the application may evolve at different places, and newer versions may break the API or semantic contract between the application and the library.

Library upgrades are inevitable for software. Sometimes the upgrades are necessitated by the discovery of security flaws. Continuing to use outdated libraries can leave applications extremely vulnerable from a security perspective. Organizations with high security requirements may even mandate requiring the security fixes where the library updates do not readily align with the

resources on every deployment target, requiring significant redevelopment. Other times, new features or a new way of realizing a functionality may lead to newer versions of a library, and not all such updates may retain backward compatibility or maintain the same assumptions about resource dependencies or usage. Library upgrade is therefore an important evolutionary pressure.

4.5.1.1 Transforming Android Code Using Program Query Language (PQL)

4.5.1.1.1 Motivation

One study of 100,000 Android apps across 5 different Android markets found that more than 60% of the sub-packages in Android apps are from third-party libraries [10]. Applications also possess dependencies on larger API frameworks, such as the Android System Development Kit (SDK) API. The backward-incompatible evolution of these APIs creates a significant burden on developers who must upgrade their application dependencies and client code or risk eventually falling out-of-step with security updates to their third-party libraries. API changes are usually documented but such information is not machine readable. Hence, the cost of making manual changes to applications is incurred repeatedly for each affected application.

We see an example of this with the Android permission changes Google introduced in October 2015 with Android 6 “Marshmallow”, API 23, which requires users to grant application permissions for some types of resources at runtime on first use, versus at installation time. These are termed dangerous permissions and include location services, which ATAKLite relies on. The changes require checking whether permission has been granted and, if not, invocation of an asynchronous call back that prompts the user. The user may deny access, in which case the application must handle a new execution path by messaging the user and shutting down or attempting to run the application in a degraded capacity. If an application attempts to use a dangerous permission before obtaining the user’s permission, a security exception is thrown.

Determining how to handle permission denials is very difficult to automate with any degree of generality; however, much of the code to find the impact points in the source code where dangerous permissions are used as well as the boilerplate code to intercept those requests and request user permission can be automated.

4.5.1.1.2 CP Description and Existing Approaches

Figure 69 shows the difference in behavior of the software application. The newer version of the library requires the application code to request, manage and handle permission in a totally different manner. In this CP, we demonstrated a pattern-query approach to automatically transform the software to handle such situations.

Existing work in this area includes srcML together with XSLT or Language Integrated Query (LINQ) framework [54]. srcML

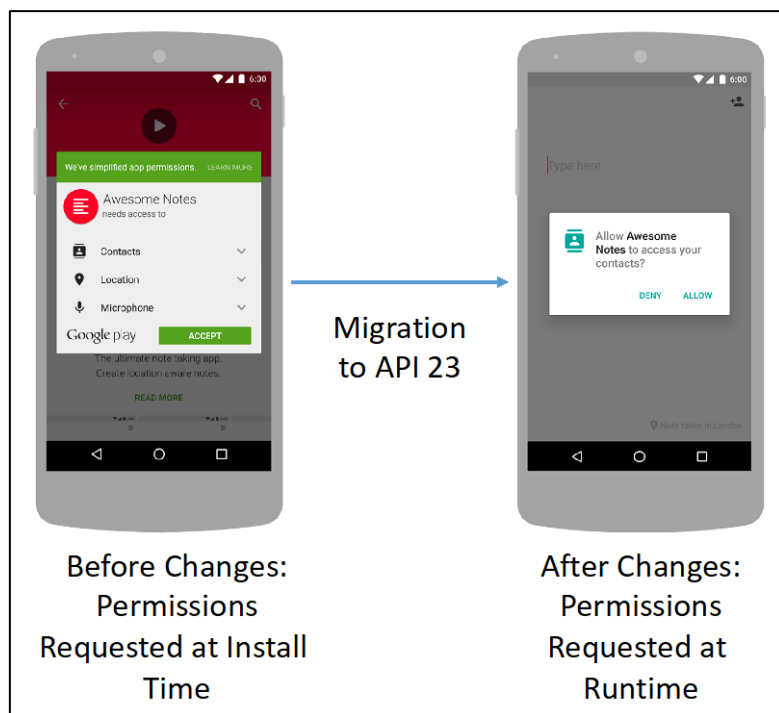


Figure 69: Android API 23 Migration

is a library that can represent source code in XML where it can be manipulated using a wealth of existing XML query and manipulation technologies, including XSLT. srcML is then used to forward-engineer source code from XML to the original programming language. While this approach is very promising, its main drawback is the difficulty of manipulating code using an XML- and XPath-based transformation language. While the researchers have extended source ML (srcML) with functions that can be invoked from XSLT to facilitate this process, it's not clear whether the approach yields significant

ease-of-use improvements as compared to writing code to directly query and manipulate parse trees (e.g., Javaparser for Java).

Use of regular expressions and related tools, such as `grep`, `sed` and `awk` may help locate potential candidate matches for simple patterns. However, they rely on character- or record-based matching which lack the expressive power to represent the structural properties of code. For example, the expression `*while.*for` could be used to match a `while` statement followed by a `for` statement; however, the expression can distinguish between a nest `for` versus an instance at the same level as the `while` statement [11]. Regular expressions also become cumbersome when the order of code statements does not matter; i.e., when matching variable declarations [11].

As an alternative, an engineer may write code that searches and manipulates the code using a syntax tree, but this is cumbersome and time consuming to code and debug. A better alternative is to design a DSL that is tailored to querying and transforming source code; hence, we introduce Program Query Language. For the Android permission problem, the source DSL program generated 91 lines of code which required a supporting environment of an additional 2,047 source lines of code. In contrast, the original DSL program contains 30 lines of DSL code + 30 lines of new or replacement Java code embedded into the DSL program. These costs are further amortized by the repeated application of the DSL program over multiple applications across industry or an organization. PQL can afford to be more compact and human readable as, unlike XSL, it does not need to conform to an XML schema.

4.5.1.1.3 Relevant IMMoRTALS Mechanisms and Capabilities

The PQL program modification¹² mechanism based on patterns to match and to modify is realized as a standalone immortals AM. PQL is an external DSL based on an ANTLR parser. PQL programs are text files that are compiled to Java code, which can be executed on a target folder to parse, query and transform all the compilation units the folders contain. The high-level toolchain architecture is shown in Figure 70.

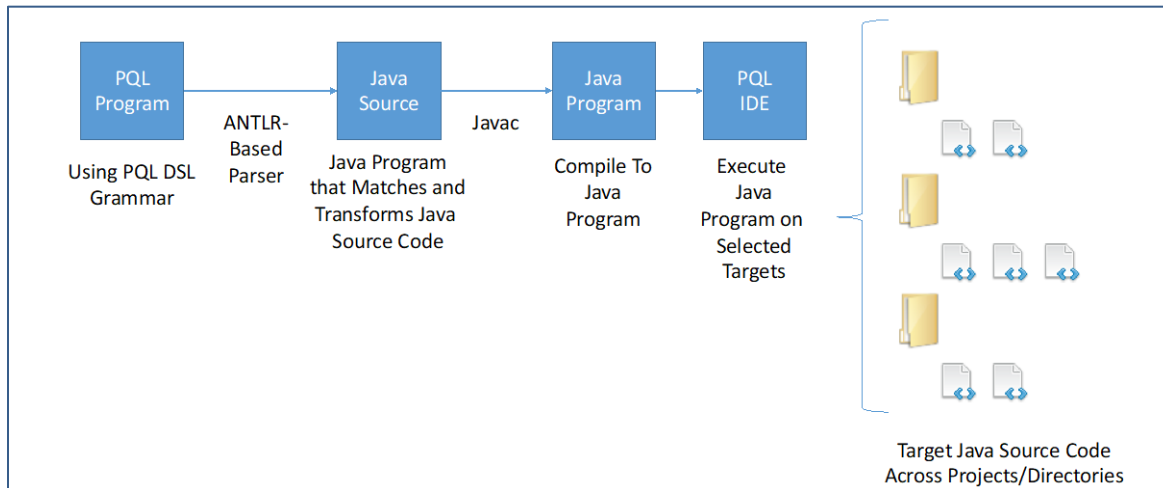


Figure 70: PQL Toolchain Architecture

The general structure of a PQL program is shown in Figure 71. Each PQL program is divided into two sections. The top half contains one or more patterns that may optionally create named bindings; e.g., if a pattern matches a variable declaration, the variable's name is placed into a PQL Context object using the binding name provided in the pattern. The binding can subsequently be referenced in the 'then' section of the program during code transformation. Code transformation is implemented as a set of specialized commands which are all executed for each match. Note that a PQL Context object is created for each instance of a pattern match.

The commands are designed to facilitate common tasks when transforming code, including adding and referencing collision resistant variable identifiers (e.g., as used in Java class fields), adding import statements to Java classes, creating new methods and replacing arbitrary code blocks. The expressivity of PQL is currently very limited as it supports the bare minimum of language concepts

¹² After the base effort, BBN was awarded a small ECP to extend PQL and apply it to a transition problem. The enhancements performed and the results of applying PQL to the chosen transition problem is described in Appendix C.

to evaluate the use of the approach on the Android permission problem for obtaining device location updates.

```
when {  
    fragment (options...) {  
        codePattern1  
        codePattern2 ...  
        //patterns create  
        //named bindings  
    }  
} then {  
    command1  
    command2 ...  
    //commands reference  
    //named bindings  
}
```

Figure 71: Structure of PQL Program

PQL patterns provide support for method invocations and arguments. As method signatures evolve, it's often the case that new parameters are introduced. In some cases, the new parameter can be mapped to an existing variable in the call site's context. Determining whether a variable is semantically appropriate for the new parameter is difficult when variables are not strongly typed or annotated. When they are strongly typed, however, it's possible to conduct a scoped search for a variable of the sought after type. This can be viewed as the reverse of the operation a symbol solver performs. PQL includes the *referenceSearch* keyword which addresses this need. The pattern results in a scoped search through all nested code blocks starting at a match site. An example is shown in Figure 72. The example triggers the PQL engine to search for a variable declaration of type `android.content.Context` that is within scope of the code pattern match named `gpsRequest`. If a match is found, the identifier's name is placed within the PQL context as

`androidContext`, the value of which can be referenced in subsequent PQL transformation commands. To implement the scoped search, we customized the Java Symbol solver integrated with Javaparser. The custom fork of Javaparser is located here (<https://github.com/peterkt2000/javaparser>).

```
referenceSearch androidContext android.content.Context gpsRequest
```

Figure 72: PQL referenceSearch Example

The complete 'when' section for the Android location permission problem is shown in Figure 73. The figure also shows the PQL IDE, which is currently a simple interface developers can use to create and compile PQL programs and review the Java source code they generate. Developers can execute the resulting program on a folder of Java source files and subsequently review which files were searched and matched by the PQL patterns, as well as the source and modified versions of each affected files.

The example shows the use of the *methodInvocationStatement*. This pattern is similar to the *referenceSearch* example: it accepts an identifier for a PQL Context binding, a Java type reference for the target object or class, as well as the method name and signature. The method arguments are also specified using a binding declaration and a type reference. This serves to disambiguate overloaded definitions.

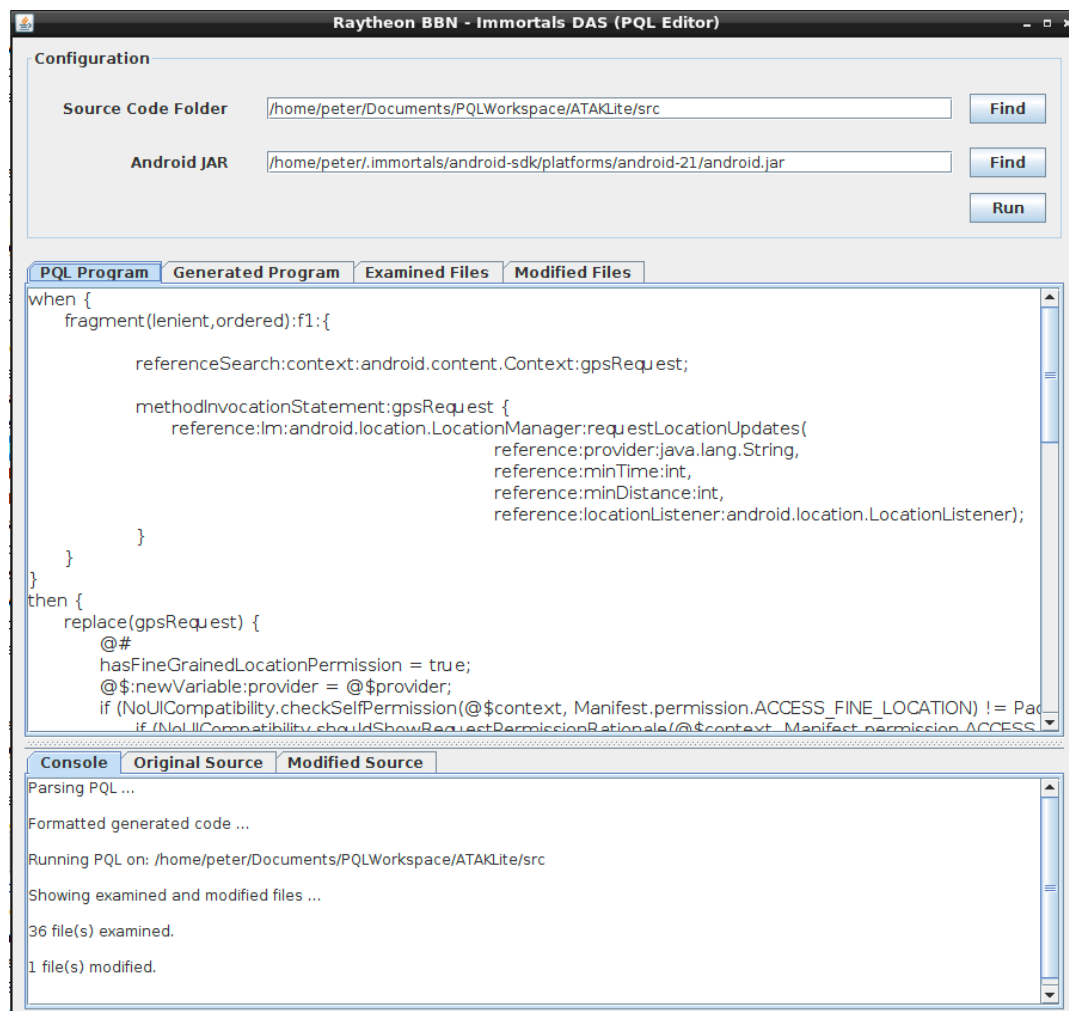


Figure 73: Simplified PQL IDE

The generated program creates an instance of the PQLEngine, which is responsible for iterating through each compilation unit in the source folder, managing the creation of PQLContext instances, and applying the fragment pattern sections. The class diagram for these and other internal helper classes are shown in Figure 74. The PQLProgramGenerator class (top left in the

figure) instantiates the PQLLexer (which is generated by ANTLR using the PQL grammar file). The resulting token stream is fed into the PQLParser to create a parse tree, which is in turn walked using the PQLListener instance. The PQLListener implements the appropriate interface that covers the production rules in PQL's grammar and emits Java source code. The 'when' section of a PQL program contains one or more fragments. Each fragment consists of one or more PQL code artifact pattern statements. Fragments allow the scoping of different pattern matching options. Two binary options are defined: lenient vs. strict and ordered vs. unordered. The first option determines whether the fragment's pattern statements should be interpreted to fully specify the code statements at a potential match site or whether they should be interpreted as partial descriptions. For example, if a fragment contains *patternX* followed by *patternY*, should the PQL engine allow other code statements not described by these two patterns to be interleaved between them in order to produce a successful match.

The second option determines whether the order of pattern statements matters within the fragment. In many cases, the order of code statements is not important for a match to be considered successful. Currently, the only option pair supported is lenient/ordered; i.e., the PQL engine will interpret pattern statements as partial descriptions of the code site but require matches to obey the order they are specified within the fragment. Note that all options are evaluated within the context of a compilation unit, which is a Java source file. Java sources files, however, may contain one or more class definitions. Additional fragments options or pattern statements may need to be introduced to allow the developer to constrain queries to a single Java class. Each fragment in the grammar is represented by a FragmentPattern Java class.

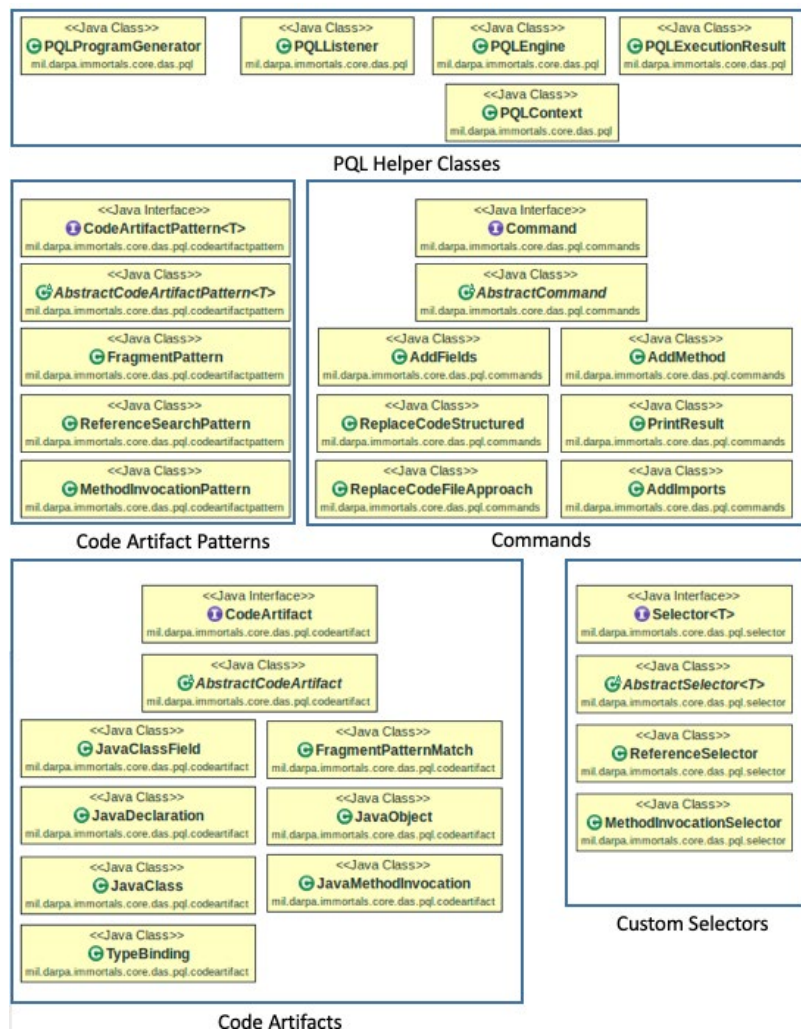


Figure 74: PQL Internal Class Diagram

The PQLEngine iterates through each compilation unit in the source tree and builds a parse tree representation using the Javaparser library. The parse tree is placed into the PQL Context. Next, the PQLEngine invokes the *match* method on each FragmentPattern instance contained in the PQL source program. Each FragmentPattern instance in turn delegates matching to the concrete CodeArtifactPattern instances it contains. Each concrete CodeArtifactPattern contains the logic necessary to inspect the parse tree for the compilation unit and assess whether a match exists. Common functionality across CodeArtifactPattern classes is encapsulated in Selector instances.

As matches are found, they are represented using concrete CodeArtifact classes, each of which contains multiple pieces of information to describe the code site where the match occurred. The CodeArtifact instances are stored in the PQLContext using the identifier provided in the pattern statement. For example, in Figure 73 we see that the methodInvocationStatement pattern command contains a binding called 'gpsRequest' (this is the second element in the command's syntax). Any matches for this pattern command are represented as a concrete CodeArtifact instance and added to the PQLContext using this name.

The current design of PQL allows the *referenceSearch* pattern to be placed anywhere in the 'when' section; e.g., it does not need to be placed after its target binding 'gpsRequest'. The PQL engine employs a multi-pass algorithm over pattern statements in order to create a more natural language for the developer. The stopping condition for the algorithm is to detect when no additional pattern statements are resolved in a given pass. This is one aspect of PQL that needs to be explored further as new pattern statements are introduced. The alternative to require the developer to order all pattern statements would also work provided pattern statements are acyclic.

```

replace(gpsRequest) {
    @#
    hasFineGrainedLocationPermission = true;
    @$:newVariable:provider = @$provider;
    if (NoUiCompatibility.checkSelfPermission(@$context, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
        if (NoUiCompatibility.shouldShowRequestPermissionRationale(@$context, Manifest.permission.ACCESS_FINE_LOCATION)) {
            System.out.println("ATAKLite requires location services.");
        } else {
            NoUiCompatibility.requestPermissions(this,
                new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
                @$:newVariable:requestCode);
        }
    } else {
        @$gpsRequest
    }
    #@
}

addMethod(gpsRequest) {
    @#
    public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {
        if (requestCode == @$:newVariable:requestCode) {
            if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                @$!m.requestLocationUpdates(@$:newVariable:provider, @$minTime, @$minDistance, @$locationListener);
            } else {
                //Permission denied
                System.out.println("Permission denied; scale down, use different location provider, exit app., etc.");
            }
            return;
        }
    }
    #@
}

```

Figure 75: PQL 'then' Section

The ‘then’ section of a PQL program contains one or more concrete Command instances, each of which contains a binding reference. The Command instance can retrieve the CodeArtifact using this binding name. This provides developers with great flexibility in the actions they perform on different patterns matched in the ‘when’ section of a PQL program. Command instances are not restricted by the FragmentPattern definitions. Each Command instance accesses and makes its modifications directly to the stored compilation unit in the PQLContext.

The final step before the PQLEngine resets the PQLContext and moves to the next compilation unit is to persist the modified parse tree to the original file.

The ‘then’ section of the PQL program is shown in Figure 75. Most of the code is Java code required to support the new Android permission model. We will focus on the ‘replace’ command, which is implemented by the ReplaceCodeStructured class shown in Figure 74. In this example, this command replaces the CodeArtifact in the PQLContext with the name ‘gpsRequest’ with the provided code between the ‘@#’ and ‘#@’ tags. This provides the developer with a simple way to provide arbitrary lines of code as a replacement for a match. Note that the replacement code contains other PQL keywords, such as ‘@\$:newVariable:provider’. This provides a reference to new class fields introduced using the ‘addFields’ command (not shown in the figure). In this example, the provider class field is assigned to another variable in the PQLContext, namely the provider binding in the MethodInvocationStatement. Note also the use of @\$context. Finally, note how the original CodeArtifact matched in the ‘when’ section is included in the replacement (in the *else* section of the *if-then* statement). The addMethod command is similar, though it makes additional use of the named bindings from the matched code (e.g., @\$minTime, @\$minDistance, etc.).

4.5.1.1.4 Evaluation

4.5.1.1.4.1 Internal Evaluation

Internal evaluation testing of PQL focused on interactive debugging and unit tests of the toolchain, including the custom javaparser tool. Interactive debugging was required to ensure that the reverse symbol solver capability that was added to javaparser worked correctly for variable definitions at different scopes from a given point of reference. These scopes included formal parameter declarations, local private fields and static declarations and public static fields in external classes. For regression testing, the existing javaparser unit tests were executed in the process of building javaparser with the reverse symbol solver capability. All tests successfully passed. Lastly, the toolchain was successfully tested in a local environment which simulated the MIT LL test harness and runtime platform.

4.5.1.1.4.2 External Evaluation by the Voice of the Offense Team (Lincoln Laboratory)

MIT Lincoln Laboratory evaluated PQL’s functionality as part of a suite of technologies for responding to third-party library changes, including performing selective updates and test-driven program minimization, both discussed in other sections of this document. The results of the evaluation are shown in Figure 76.

The application under test for the PQL-based CP was the ATAQLite application developed in Phase 1, in particular its use of GPS location services. If the MIT test harness indicated that a simulated Android upgrade was required, the DAS executed the PQL program against the ATAQLite client and made the necessary repairs. A set of 16 regression tests are executed to verify that the ATAQLite clients can receive and transmit location information correctly after the adaptation. The baseline, perturbed and adapted flags for Android23 correspond to the results for this specific CP in Figure 76. In the baseline case, all the validation tests passed, whereas when perturbed only 80% of them passed. But with PQL, the adapted software passed all tests just as in the baseline case—this implied 100% intent preservation through PQL adaptation.

It was not feasible to actually update ATAQLite SDK during evaluation. Hence, code changes were made to ATAQLite to set a flag if validation was performed. The value of this flag was verified at run time and, if set to false, the application generated a runtime security exception. Similarly, since it was not possible to prompt a user for permission during the automated evaluation, a user response was simulated and set to allow access.

To repair the ATAQLite application, over 36 compilation units containing 2,464 source lines of code were examined. The search pattern included a method invocation and 6 declaration statements across every declaration context visible to the method invocation. There were 20 lines of modified and new code in the pattern match.

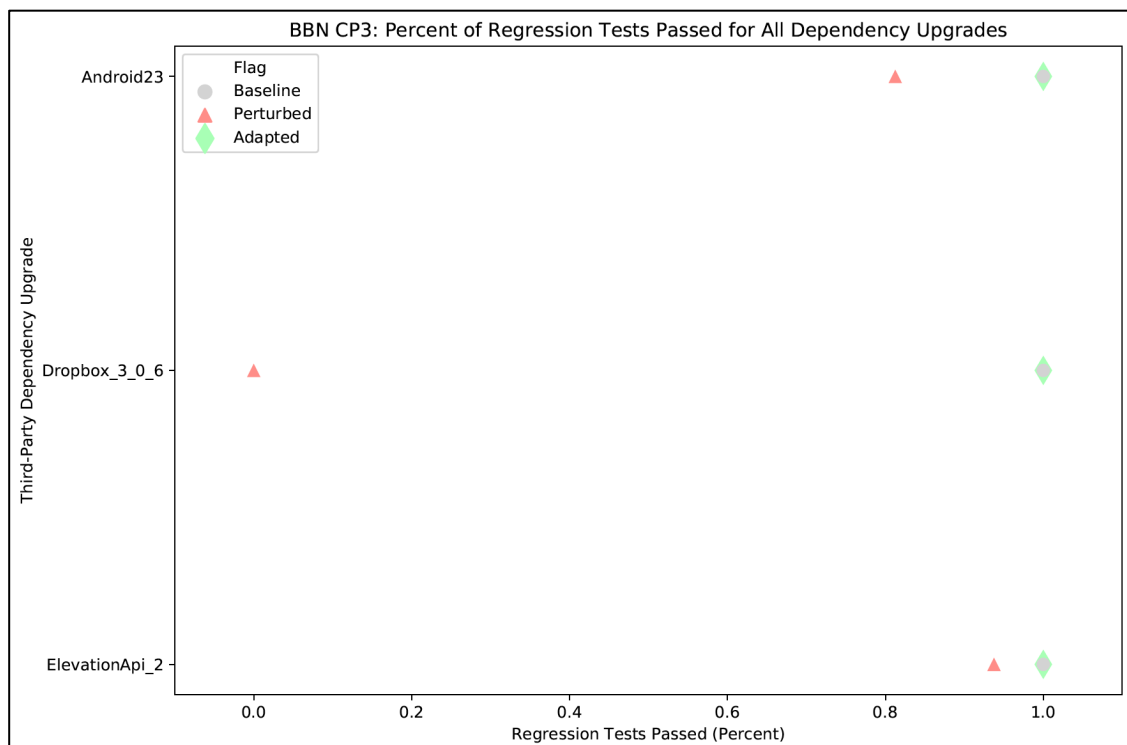


Figure 76: Third-Party Library Evolution Tests

4.5.1.2 Hierarchical Delta Debugging Based Resource Adaptation

4.5.1.2.1 Motivation

A wide variety of data is utilized by situational awareness (SA) applications like ATAK. The core purpose of such applications is to share information, but that information often needs additional context in order to be useful. The additional context is usually provided by different sources such as a map provider providing maps, a navigation service providing navigation tailored to current traffic conditions, a weather service, or an elevation details provider, on top of a default level of information provided by the original data source. For a given deployment, not all contextual information may be relevant—but typically the application is programmed to handle as many detailed contexts as the original designers had specified. The sources of some of these additional contexts can be local (e.g., cached maps), or over the network (e.g., going to a remote database), whereas for some, going remote is the only possibility (e.g., navigation based on current traffic). In many cases accessing such context information is handled by third party libraries, and it is not uncommon that different versions of such libraries change how they access and provide the data to the consuming application. Such changes are an important evolutionary pressure for the consuming application since going from local to remote or changing fidelity of the data can break the application and/or fail to meet the application requirements.

4.5.1.2.2 CP Description

In this CP we focus on the Elevation library that is used by our Android based SA application. Enhanced elevation data provided by this library is used to augment spatial SA. A new version of the library is published and the client application is required by policy to use the latest version of the library. However, the current version of the library uses an internal database to get enhanced elevation information while the updated version uses a network call. And most importantly, the current deployment environment prevents outbound network connections.

We assume that the new version does not modify the API, and the application compiles readily against the new version. The goal of this CP is to apply hierarchical delta debugging based resource adaptation to find a minimized program that eliminates unnecessary resource usage and dependency so that there is no surprise at runtime.

In this CP, data from the calls to the elevation library were overwriting the default elevation data obtained from the GPS system. If the enhanced data is not really needed for a mission, then it should be possible to eliminate library calls that results in failure due to lack of network connectivity to create an application that is custom adapted for that mission.

The pattern of overwriting default values with enhanced values is not uncommon—developers/vendors often add these enhancements to improve the user experience, or quality of the data as an up sale even when in many use cases and users really do not need or care about the enhancements. However, as this CP illustrates, these decisions introduce dependencies that may be invalid in specific deployment environments.

4.5.1.2.3 Relevant IMMORTALS Mechanisms and Capabilities

Following the DAS adaptation workflow, when the change (mandatory use of the new Elevation library) is pushed to the DAS, the CM is used to build the application with the new library and the VM is used to perform the validation tests. The application is built successfully since the library upgrade did not involve any API changes. But two tests failed: An enhanced elevation test, and a test that utilizes the overall SA data and requires a valid elevation value (which is provided at a much lower precision by the GPS). Under the assumption that new library provides unneeded functionality violating resource constraints, a custom AM that realizes hierarchical delta debugging based resource adaptation (hddRA) then attempts to eliminate the unnecessary resource usage and dependencies.

Unlike other AMs, the hddRA technique relies heavily on multiple quick iterations of test execution. To facilitate quick turnaround and automation, we forked the ATAKLite application into an alternative CLITAK, which is essentially the same application, but with the Android API usage substituted for Java API usage to get around the speed and stability issues we have encountered using Android emulators.

4.5.1.2.3.1 Relevant Knowledge Builders

The hddRA-based Adaptation Module depends on the following metadata about the application:

- How to build the application
- How to execute specific tests
- Specific tests applicable to the application

As explained earlier in Section 4.1, most of this information is derived during the application ingestion process. In addition, hddRA requires knowledge about which tests are critical for a given configuration and which are not. In this CP we manually labeled the tests that could be *sacrificial* for the given deployment, and the AM treat the rest of the applicable tests to be critical i.e., in order for the minimized program to be valid, all the unlabeled tests must pass.

Given the labeled tests (i.e., tests that can be sacrificed), a specialized KB was used to derive the following information:

- A list of prioritized classes targeted for minimization

The KB determined this information by executing the common code coverage tool JaCoCo on the failing sacrificial class to determine what classes were involved in execution of the failing test case, and prioritizing them by coverage. This list of prioritized classes was used as a generic heuristic to focus on first shedding code from classes that are less likely to invalidate critical tests.

4.5.1.2.3.2 hDDRA-based Adaptation Module

The basic idea behind hddRA is to remove code that is considered sacrificial based on their lack of coverage in the unlabeled (i.e., non-sacrificial or critical) tests, and thereby minimizing the application's resource usage. This Test-Based Software Modification (TBSM) concept is notionally depicted in Figure 77. A modified version of hierarchical delta debugging is used to guide which lines can safely be removed.

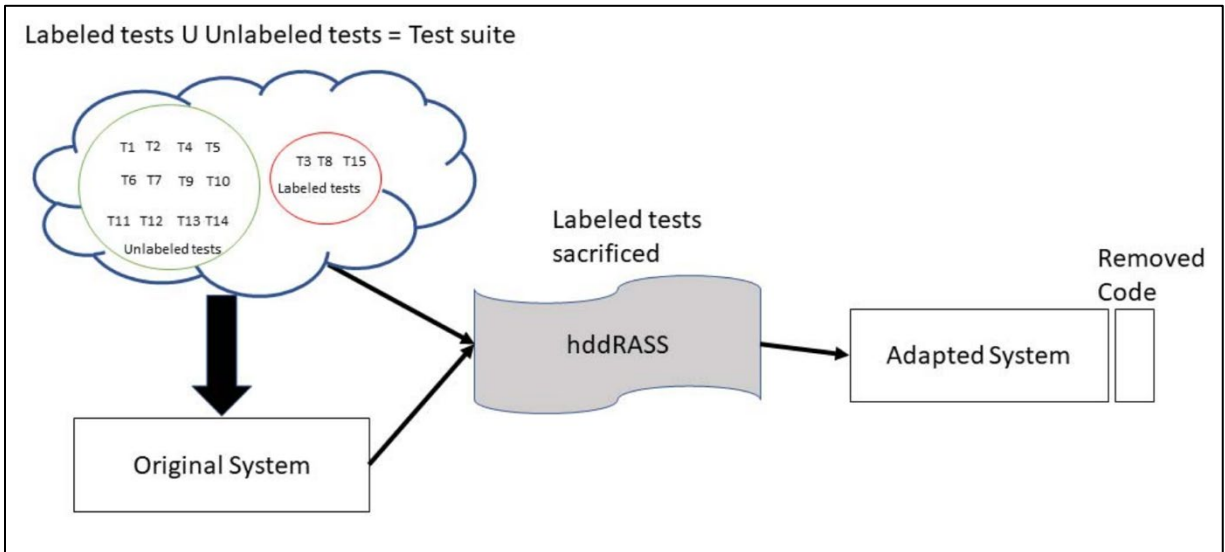


Figure 77: Schematic depiction of test based minimization realized by hddRA

Delta-debugging [32] is a binary-search like algorithm intended to reduce the size of failing test cases. It removes components of a test that are not required in order for the test to fail. Hierarchical-delta-debugging (HDD) [33] modified the algorithm to produce better results for tree-structured test inputs, especially, e.g., programs input to compilers. Cause reduction [34], [35] further proposed that delta-debugging/HDD are not limited to reducing tests with respect to the property “the test fails” – they can also reduce a test so that it is shorter but covers the same code, uses the same amount of memory, and so forth. Along this line, hddRA was used to reduce actual programs, resulting in a reduced resource footprint [27]. Our modified hierarchical delta debugging technique combines hierarchical delta debugging with statement deletion mutation. The program is reduced until a locally minimal version of the program is found that can still pass all the unlabeled tests.

Due the usage of HDD, hddRA is still a greedy search-based algorithm, with similar worst-case time complexity $O(n^3)$ [27]. This cost is even more significant with hddRA, as every attempted reduction step consists of compiling a new version of the program and running potentially most of the program’s test suite, which led to further research to reduce the search space defining three heuristics to reduce the number of unnecessary test iterations [28]:

- H1: Only attempt to remove simple statements as defined by *java.parser.ast.statement*
- H2: Only attempt to remove *IF*, *RETURN*, and *EXPRESSION* statements
- H3: Only attempt to remove statements covered by Coverage Based Likely-removable Statements (CBLS), where CBLS consists of statements that are both covered by the removed tests and not covered by the required tests

By using these heuristics to select statements to be removed, the hddRA-based AM achieved a significant reduction in search time while maintaining close to the accuracy of an adaptation

considering all statements. This was again iterated on by applying CBLS (Heuristic 3) to existing Fault Localization (FL) strategies in a strategy called Adaptive Fault Localization, or AdFL.

4.5.1.2.4 Evaluation

4.5.1.2.4.1 Internal Evaluation

Internal evaluation focused on evaluating hddRA and the underlying heuristics independently of the DAS

Evaluation of Custom Heuristics: These results cover the execution of hddRASS using six randomly chosen classes [28]. They were executed on test suites where 10% (L10) and 20% (L20) of the tests were considered sacrificial and removed from the set of tests that were validated against.

The efficiency gains can be seen in Table 20, showing significant increases in efficiency for the tested classes compared to execution without heuristics. The accuracy can be seen in Table 21 showing decent accuracy, with perfect accuracy for Heuristic 3, CBLS. Accuracy is defined as the percentage of statements removed in comparison to the baseline without heuristics.

AdFL: The Adaptive Fault Localization strategy

applied Heuristic 3, CBLS to the existing Fault Localization (FL) techniques Tarantula [30], Ochiai, Op2, and DStar [31]. This was applied to the Elevation API used for IMMoRTALS as well as the removal of undo/redo functionality within the NetBeans IDE [29][28]. The improvement in efficiency when AdFL is used in comparison CBLS can be seen in Table 22 where the EXAM SCORE [36] is being used as the effectiveness benchmark.

Table 20: Efficiency gained by heuristics (as %) broken down by omitted test percentage for H3

Subject	H1	H2	H3-L%10	H3-L%20
Array2DRowRealMatrix	16	22	63	65
EigenDecomposition	27	19	26	33
MillerUpdatingRegression	29	12	51	15
SevenZOutputFile	15	11	24	23
ZipFile	6	9	16	20
ZipArchiveEntry	11	10	47	36
MEAN	17.33	13.83	37.83	32

Table 21: Heuristic Accuracy

Subject	H1		H2		H3	
	L10	L20	L10	L20	L10	L20
Array2DRowRealMatrix	100	100	83	85	100	100
EigenDecomposition	100	100	75	85	100	100
MillerUpdatingRegression	85	89	85	85	100	100
SevenZOutputFile	100	100	86	88	100	100
ZipFile	89	92	66	73	100	100
ZipArchiveEntry	100	75	75	90	100	100
MEAN	96.25	97.25	85.75	88.75	100	100

Table 22: Percent improvement achieved by AdFL

Application	CBLS	Tarantula	Ochiai	Op2	Barinel	DStar	AvgAdFL	WorstAdFL	AdFL vs. CBLS
Elevation-API	55.17	91.72	93.79	90.34	91.72	92.41	92	90.34	66.75
NetBeans IDE undo-redo	90.71	92.28	93.42	96	93.85	94.42	94	92.28	3.62

4.5.1.2.4.2 External Evaluation by the Voice of the Offense Team (Lincoln Laboratory)

This evaluation narrowly focused on intent-preservation – passing all critical tests after the new version of the library was integrated. The evaluated AM was configured to use hddRA with the prioritized list of classes; custom heuristics and AdFL was not used. Also, the DAS was configured to stop as soon as the unlabeled tests pass, and was not ran to maximize code minimization or minimize resource usage.

The baseline, perturbed and adapted flags for ElevationAPI_2 correspond to the results for this specific CP in Figure 76. All unlabeled tests passed in the baseline, whereas 95% of them passed when perturbed. With hddRA, the adapted software passed all tests again.

4.5.1.3 Selective Library Update

4.5.1.3.1 Motivation

Third-party libraries (TPL) have been extensively used in Android to provide rich complementary functionalities for Android apps and ease the app development. Despite the benefits, TPLs can bring serious security problems for Android app. It has been revealed [1] that 70.40% of Android apps include at least one outdated TPL and 77% of the app developers only update at most a strict subset of their included TPLs, leaving many known and easy-to-exploit security vulnerabilities unpatched. In fact, updating TPLs in Android apps can be so time-consuming and tedious that developers often leave them outdated.

4.5.1.3.2 CP Description

This CP involves our Android client application modified to use a third party library like DropBox, and artificially forcing it to not use the latest version of the DropBox library as a whole. However, when new versions of the DropBox library appears with security patches, the Android application is required to apply the security patches. This creates an adaptation pressure where the developers need a way to apply just the security fixes to the existing version of the DropBox library.

Tests showing the use of DropBox and exploiting the vulnerabilities in the DropBox library were also developed for this CP.

4.5.1.3.3 Relevant IMMORTALS Mechanisms and Capabilities

To solve the aforementioned problem, we aim to automatically generate updates for TPLs in Android apps such that it does not require any code modification on the app side and introduces no impact to the library interactions with other components locally and remotely. To this end, we design and implement a prototype called LIBBANDAID. Figure 78 shows the overview of LIBBANDAID. There are four major components in LIBBANDAID: preprocessing, diffing analysis, update generation, and selective updating. The four components are introduced as follows:

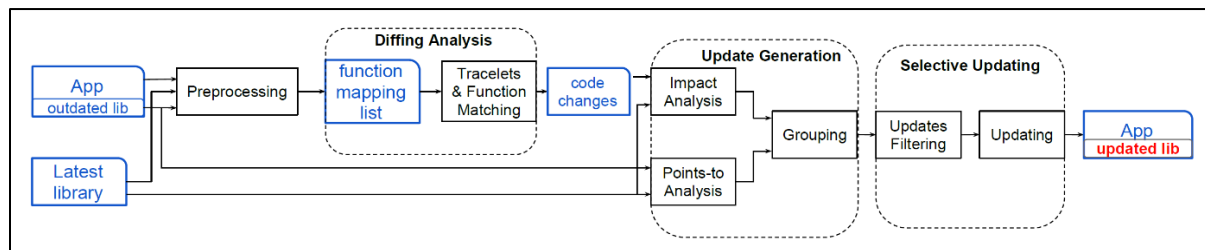


Figure 78: Architecture overview of LIBBANDAID for performing selective library update

The LIBBANDAID tool was incorporated in an IMMORTALS Adaptation Module for this CP. This AM can run standalone, in the DAS it was invoked from the DAS's adaptation workflow. The DAS also used the generic KB, CM and VMs to derive metadata about tests application build processes and running tests and collecting data to determine adaptation validity. The remainder of this section will focus on the technical details of LIBBANDAID.

4.5.1.3.3.1 Preprocessing

This step is to filter out the unchanged functions and generate function pairs that are modified across the two versions. Preprocessing components takes as input an app with outdated library and a latest version of the library, and outputs a set of function pairs. More specifically, it extracts the outdated library within the given app, analyzes all classes in the two versions of the library and performs function level byte-by-byte comparisons.

4.5.1.3.3.2 Diffing Analysis

Diffing analysis in LIBBANDAID is to perform function level matching with a granularity of code statement so as to comprehend the exact code changes between old and new versions of a given library. To achieve this goal, we leverage the Tracelet Execution [2] and use 3-tracelet to perform code matching at code statement level. Given the output of preprocessing, 3-tracelets are generated to capture partial flow information by breaking down the control-flow graphs for each function pair. Then, the distance between tracelets are calculated to match code statements.

4.5.1.3.3.3 Update Generation

Once LIBBANDAID identifies all the code changes between the old and new versions, it starts the update generation process. The whole process takes three inputs: 1). code changes generated by diffing analysis; 2). the old version of the library; and 3). the new version of the library, and generates one output (a set of updates). It first generates system dependence graphs (SDGs) for new and old library, and then generates a slice for each code change by performing impact analysis. Finally, it performs grouping based on the alias information gathered from points-to analysis to produce updates.

4.5.1.3.3.4 Selective Update

The last component of LIBBANDAID is selective updating. It takes the updates generated in the previous step, performs filtering to discard the ones that could potentially break the non-intrusiveness, and eventually updates the old library to generate a new app with an update library. The core part of this step is to systematically devise a set of pre-defined rules for filtering so that the non-intrusiveness of our generated updates can be preserved.

As discussed earlier, the key component of LIBBANDAID is the **slicing algorithm** in the **Update Generation** module. We propose a *Value-sensitive Differential Slicing* algorithm to generate a slice for each code change generated by diffing analysis. The details of the Value-sensitive Differential Slicing algorithm are shown in Figure 79, which is a dependence graph based slicing algorithm.

As shown in Figure 79, the algorithm first locates the *diff* in two SDGs (line 6) and adds *stmt_n* into a *workingSet* (line 7) to start the iterative process. The algorithm will continue running as long as the *workingSet* is not empty (line 9). For every statement in the working set, we extract its immediate successors in SDG (line 11). For every immediate successor *succ*, the algorithm checks

if it is another code change. There exist two cases under this scenario. First, if *succ* is a code that contains a new function invocation, our algorithm needs to leverage traditional slicing by calling *Forward_Slicing()* to keep track of the new function call (lines 13-14) as all its codes are new codes compared to the old version. Second, if *succ* is a normal code change, we consider it as another input to a recursive function call for *V_Slicing()* (lines 15-16).

When *succ* is not a code change, we add it into the *workingSet* as well as the slice if it is only control-dependent on *stmt* (lines 17-19). When *succ* is a return statement, we apply the second optimization by performing function-wise VSA (Value-Set Analysis) [3] for all return statements to improve the accuracy (lines 20-23). When *succ* is data-dependent on *stmt*, we calculate and compare the value-sets by calling *VSA()* to extract value formulas for both old and new versions and only add *succ* when *stmt* has impact on it (lines 25-30). Eventually, it produces a slice by returning *slice* (line 35).

Algorithm 1 Value-sensitive Differential Slicing

```

1: input1:  $diff \leftarrow \{stmt_o, stmt_n\}$ 
2: input2:  $SDG_n \leftarrow \{\text{SDG of the new library.}\}$ 
3: input3:  $SDG_o \leftarrow \{\text{SDG of the old library.}\}$ 
4: procedure  $V\_Slicing(diff, SDG_n, SDG_o)$ 
5:    $slice \leftarrow \emptyset$ 
6:    $f_n \leftarrow Locate(stmt_n, SDG_n); f_o \leftarrow Locate(stmt_o, SDG_o)$ 
7:    $workingSet \leftarrow workingSet \cup stmt_n$ 
8:    $slice \leftarrow slice \cup stmt_n$ 
9:   while  $workingSet \neq \emptyset$  do
10:     $stmt \leftarrow workingSet.remove()$ 
11:     $Set_{succs} \leftarrow ImmediateSuccessors(stmt, SDG_n)$ 
12:    for  $succ \in Set_{succs}$  do
13:      if succ contains new invocation then
14:         $slice \cup \leftarrow Forward\_Slicing(succ, SDG_n)$ 
15:      else if succ is another  $diff'$  then
16:         $slice \cup \leftarrow V\_Slicing(diff', SDG_n, SDG_o)$ 
17:      else if succ is control-dependent on stmt then
18:         $slice \leftarrow slice \cup succ$ 
19:         $workingSet \leftarrow workingSet \cup succ$ 
20:      else if succ is return statement then
21:        if  $!(RetVS(f_o) \subseteq RetVS(f_n))$  then
22:           $slice \leftarrow slice \cup succ$ 
23:           $workingSet \leftarrow workingSet \cup succ$ 
24:        end if
25:      else if succ is only data-dependent on stmt then
26:         $vf_n \leftarrow VSA(succ, slice, SDG_n)$ 
27:         $vf_o \leftarrow VSA(succ', slice, SDG_o)$ 
28:        if  $!(vf_n \subseteq vf_o)$  then
29:           $slice \leftarrow slice \cup succ$ 
30:           $workingSet \leftarrow workingSet \cup succ$ 
31:        end if
32:      end if
33:    end for
34:  end while
35:  Return slice
36: end procedure

```

Figure 79: Value-sensitive Differential Slicing algorithm

4.5.1.3.4 Evaluation

4.5.1.3.4.1.1 Internal Evaluation

We collected nine popular Android third-party libraries including Butterknife, Dropbox, EventBus, Glide, Gson, Leakcanary, Okhttp, Picasso, and Retrofit, with a total of 173 security commits over 83 different versions to evaluate our system.

We first collected ground truth based on commit information in GitHub repositories to gather the vulnerability information for all the 173 security commits. Then, we compiled libraries into a number of testing versions with two requirements: (1) each testing version contains at least one security commit; (2) these testing versions cover all the security commits. Finally, we developed Android apps that utilize these testing versions. For each testing version other than the latest one, we fed the Android apps with these versions along with the latest version of each library into LIBBANDAID for evaluation. For instance, Butterknife library has 6 security commits from version 7.0.1 to 8.0.1. We compiled 6 testing version v1 to v6 to guarantee each one will contain at least one commit. Then we developed five Android apps a1 to a5 that use testing versions v1 to v5 and feed (a1,v6), (a2,v6), ..., (a5,v6) into LIBBANDAID for experiments.

Security commits can be divided into three categories: (1) *patched*; (2) *fail to patch*; and (3) *potentially patchable*. *Patched* means our system can successfully update the library with the commit. *Fail to patch* shows the commits that are filtered out because of the violation of our pre-defined rules. *Potentially patchable* indicates the commits that change the APIs of the library.

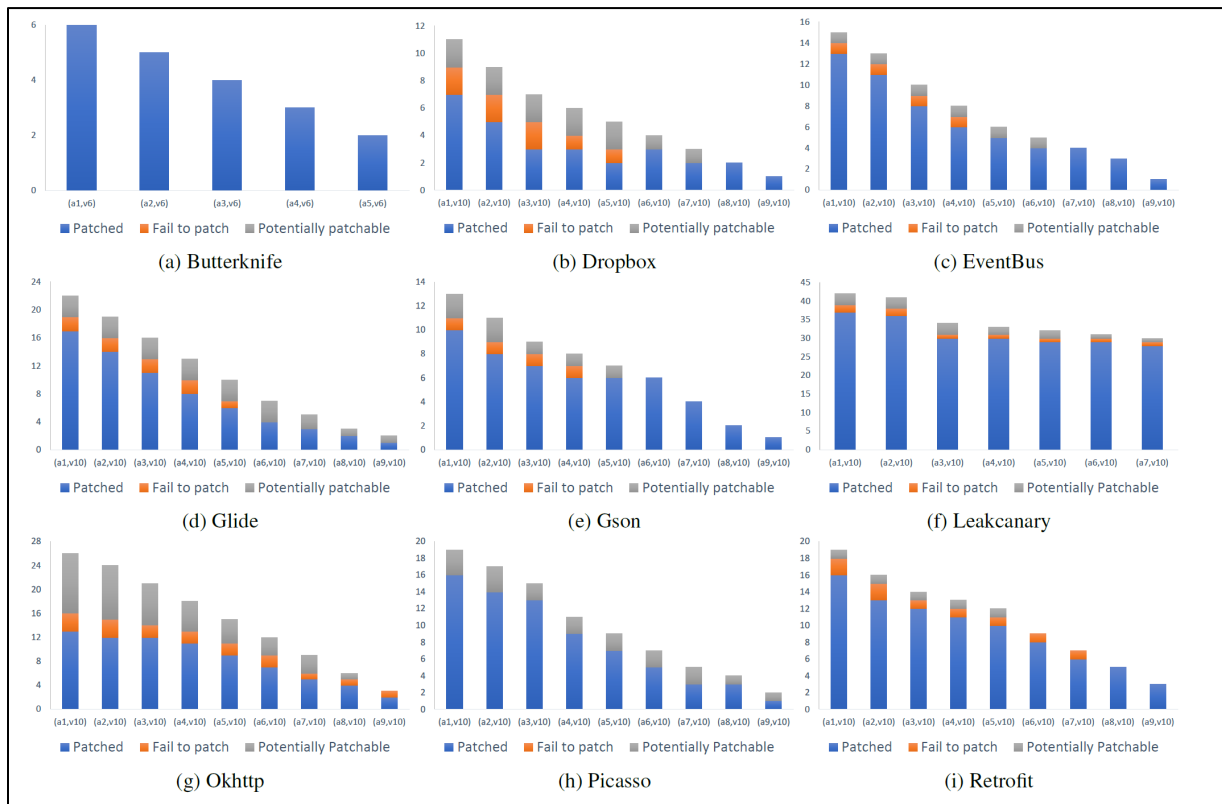


Figure 80: Results of experiments on nine Android TPLs

LIBBANDAID may still update the *potentially patchable* one as long as the analyzed Android apps do not directly invoke the changed APIs.

Figure 80 gives the results of these experiments in absolute numbers for the 9 libraries. The x-axis shows each execution of LIBBANDAID while y-axis is the absolute number of vulnerabilities. For example, the x-axis in Figure 80 (b) gives the nine executions from (a1,v10) to (a9,v10) for Dropbox library and the y-axis shows the number of security commits updated for each run. By looking at the first bar in the figure, we can see that there are 11 vulnerabilities between the old the new versions of the library. LIBBANDAID was able to fix 7 of them but fails in 2. Moreover, there are 2 security commits that change the APIs, so we mark them as *potentially patchable*. In a word, from the experiments, LIBBANDAID could achieve an average success rate of 80.6% for updating security commits and even higher rate of 94.07% when combining with the *potentially patchable*.

4.5.1.3.4.1.2 External Evaluation by the Voice of the Offense Team (Lincoln Laboratory)

The external evaluation of this CP focused on the ATAKLite application using DropBox. The baseline, perturbed and adapted flags for DropBox correspond to the results for this specific CP in Figure 76.

For the DropBox security updates considered in this evaluation, IMMoRTALS with LIBBANDAID –based AM was able to patch the baseline library in such a way that the exploits did not work, and at the same time no baseline tests were broken.

4.5.2 Adapting ROS1 Code to ROS2 - a Machine Translation Approach (Phase 3, CP 3)

4.5.2.1 Motivation

In this section, we introduce the learning-based approach for the ROS1 to ROS2 migration problem. ROS2, which leverages ROS-SE (ROS Secure), has more security features than ROS1. However, it is tedious and error-prone to manually migrate packages from ROS1 to ROS2, thus automatic transformation framework is required.

The nature of migrating ROS1 code to ROS2 bears considerable similarity to language transation—each ROS version has its own guiding rules like language grammar and idioms, and ROS2 rules can be used to convey the essence of the ROS1 code. Recent successes in use of automated machine learning (ML) for machine translation motivated us to research a ML based Adaptation Module for this problem. For the automatic learning based approach, we mainly face the following challenges:

1. New APIs are added to ROS2, and existing APIs also change dramatically;
2. There are only a limited number of migration examples available;
3. Code commits of ROS2 also contain irrelevant changes.

To mitigate these challenges, we used the open-source AST matching tool GumTree [4] to map the migrated statements. We also conducted extensive hyper-parameter tuning to fine-tune the machine learning and deep learning models to get the best performance.

4.5.2.2 Relevant IMMoRTALS Mechanisms and Capabilities

The learned-based technique is developed as a standalone mechanism, and does not depend on the software metadata that the IMMoRTALS KBs produce. Instead, the pipeline embedded in the standalone mechanism performs its own discovery and learning. The current state of the learning mechanism is not able to produce compilable code in a fully automated manner—therefore the integration of the AM that wraps this mechanism is also partial. As the technology matures, and is able to produce compilable ROS2 translation, the DAS should be able to leverage its build and validation capability toward a fuller integration. In the rest of this section we describe the details of the emerging learning-based mechanism for code migration.

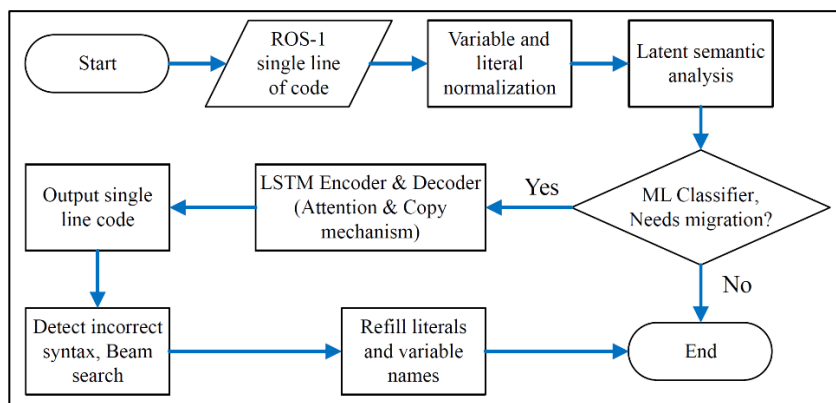


Figure 81: Framework of learning based approach for migrating ROS1 code to ROS2 code

The framework of learning-based approach for migrating ROS1 code to ROS2 is shown in Figure 81. As shown in Figure 81, the approach mainly uses machine learning and deep learning models to automatically migrate ROS1 CPP source code to its corresponding ROS2 version. The various steps performed by this mechanism are described next.

4.5.2.2.1 Training Dataset Generation

We used the open-source tool GumTree [4] to map the cpp files' ROS1 and ROS2 versions. Based on the mappings, a training dataset was generated. Table 23 shows the statistics on the generated dataset. The following training and testing experiments are all conducted based on this dataset, using cross-validation.

Table 23: Statistics on the generated ROS1 and ROS2 migration dataset

	ROS1 CPP Files	ROS2 CPP Files
Number of Files	48	48
Number of Functions	252	259
Lines of Code	10,824	11,171
Mapped Statement Pairs	4,521	
Migrated Statement Pairs	1,018	

4.5.2.2.2 Change Prediction based LSA and Machine Learning

For a single line of code in ROS1 version, firstly the approach performs variable and literal normalization. After this, the processed line of code is fed into the “Latent semantic analysis” module, in which the Latent Semantic Analysis (LSA) [5] (also called Latent Semantic Indexing, or LSI) technique is used to extract the latent information contained in this line of code. LSA is a widely-used document vectorization technique in NLP and has been successfully used in software Change Impact Analysis [6]. In this project, LSA is used in single line code migration for the first time.

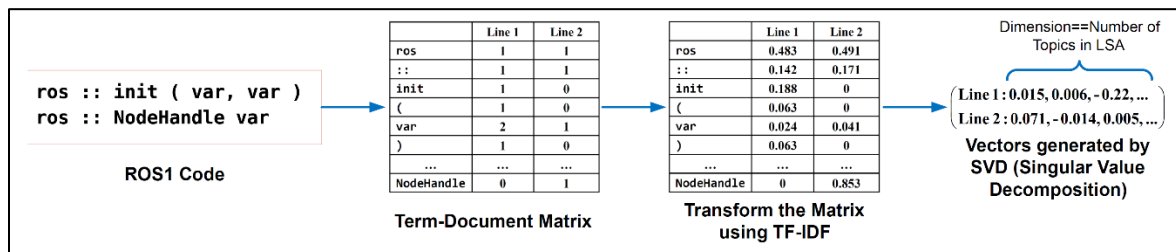


Figure 82: Calculation process of LSA for ROS migration prediction

Figure 82 shows the detailed calculation process of LSA. It can be noticed that the dimension of the generated vectors is equal to the pre-defined number of topics in LSA, which is a hyper-parameter of LSA.

After generating the vectors for each line of code, a machine learning classifier can be trained to decide whether the specific line of code needs to be migrated, as shown in Figure 81. To select the proper machine learning classifier and choose the hyper-parameter of LSA, an experiment is conducted based on ROS1 and ROS2 source code pairs. In the experimental results part of this section, we will report the results.

4.5.2.2.3 ROS1-ROS2 Translation Using Sequence-to-Sequence with Attention and Copy

After using the machine learning classifier to decide whether the specific line of code needs to be migrated, and if classifier has positive prediction, then the corresponding lines are send to a Sequence-to-Sequence [7] model to automatically generate the migration suggestions, as shown in Figure 81.

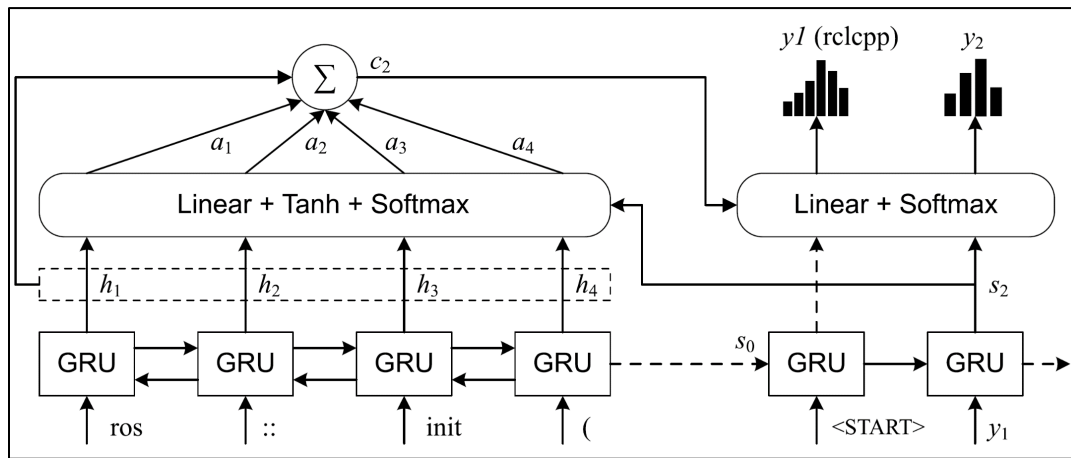


Figure 83: Network architecture of the Sequence-to-Sequence model

The network architecture of the Sequence-to-Sequence model [7] is shown in Figure 83. The Sequence-to-Sequence model also incorporates the attention [7] and copy mechanisms [8] for ROS migration. As shown in Figure 83, attention mechanism allows each output token (i.e., y_2 in Figure 83) to pay attention to different encoder hidden states when predicting a token from the vocabulary. Attention mechanism is the underlying mechanism of the state-of-the-art NLP model, which is called Transformer [9].

As shown in Figure 83, the model uses two distinct Recurrent Neural Networks (RNNs) as encoder and decoder, respectively. The input of the encoder and the decoder is firstly mapped to word embeddings by a shared embedding layer. Given a source sequence w , at time step i , the encoder

calculates a hidden state $\mathbf{h}_i$ based on the word embedding $\mathbf{x}_i$ of w_i and the previous hidden state $\mathbf{h}_{i-1}$ using its RNN. The last hidden state $\mathbf{h}_{|w|}$ is regarded as the intermediate representation of the source sequence and input to the decoder as the initial hidden state.

At decoding step j , the decoder computes a hidden state $\mathbf{s}_j$ from $\hat{\mathbf{x}}_j$, which is the embedding of the previous reference token when training or the previously generated token when testing. The model leverages the attention mechanism, so the decoder also calculates a context vector $\mathbf{c}_j$, as follows:

$$\begin{aligned} e_i^j &= \mathbf{v}^e \tanh(\mathbf{W}_h \mathbf{h}_i + \mathbf{W}_s \mathbf{s}_j + \mathbf{b}_e) \\ \mathbf{a}^j &= \text{softmax}(\mathbf{e}^j) \\ \mathbf{c}_j &= \sum_i^{|x|} \mathbf{a}_i^j \mathbf{h}_i \end{aligned}$$

where $\mathbf{W}_h$, $\mathbf{W}_s$ and $\mathbf{b}_e$ are learnable parameters and e_i^j is the score of $\mathbf{x}_i$ at decoding step j . $\mathbf{a}^j$ is the attention distribution, which informs the decoder of the importance of each encoding step. $\mathbf{c}_j$ is computed as the weighted sum of all encoder hidden states, which can be regarded as the representation of the source sequence at decoding step j .

Then, $\mathbf{c}_j$ is concatenated with decoder hidden state $\mathbf{s}_j$ to produce the vocabulary distribution P_{vocab} :

$$P_{vocab} = \text{softmax}(\mathbf{V}'(\mathbf{V}[\mathbf{s}_j, \mathbf{c}_j] + \mathbf{b}) + \mathbf{b}') \quad \text{Eq: 5}$$

where $\mathbf{V}'$, $\mathbf{V}$, $\mathbf{b}$ and $\mathbf{b}'$ are learnable parameters. P_{vocab} is used to decide which token in the vocabulary should be output at the current decoding step. It also provides the conditional probability of generating the j_{th} reference token y_j , which is:

$$p(y_j | \hat{y}_0, \dots, \hat{y}_{j-1}, \mathbf{w}) = P_{vocab}(y_j) \quad \text{Eq: 6}$$

Where $\hat{\mathbf{y}}$ is the input of the decoder.

At each training iteration, the optimization objective of the Sequence-to-Sequence model is to minimize the negative log-likelihood of the reference sequence, as follows:

$$loss = -\frac{1}{|y|} \sum_{j=1}^{|y|} \log p(y_j | \hat{y}_0, \dots, \hat{y}_{j-1}, \mathbf{w}) \quad \text{Eq: 7}$$

4.5.2.3 Evaluation

4.5.2.3.1 Internal Evaluation

In this part, firstly the experimental results of the machine learning classifier are discussed, followed by the results of the Sequence-to-Sequence model.

As discussed earlier, we should choose the machine learning classifier and the hyper-parameter of LSA. An experiment containing 4,521 ROS1 and ROS2 statements (including 1,018 positive pairs) has been conducted. We used the standard 10-fold cross-validation in this experiment. Figure 84 shows the performances (in terms of F-measure) of different classifiers with different number of topics in LSA. As shown in Figure 84, we have evaluated three different machine learning classifiers: Support Vector Machine, Random Forest, and K-Nearest Neighbors. It can be observed that when the Support Vector Machine are used and the number of topics in LSA are 768, we can get the best performance, for which on average F-measure is 0.937 and AUC is 0.987. Such results signify that the machine learning classifier can achieve a precise prediction for single line ROS migration.

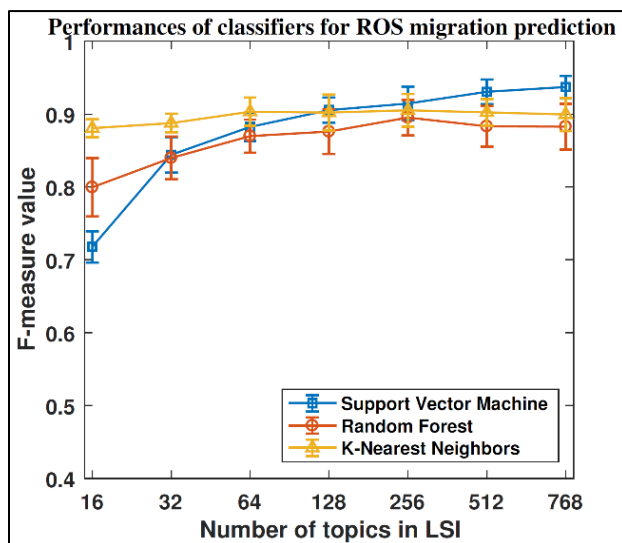


Figure 84: Performances of different classifiers for ROS migration prediction

We performed extensive hyper-parameter searching by testing various configurations of the Sequence-to-Sequence model. The configurations tested different combinations of RNN Cells (Long Short Term Memory or LSTM, and Gated Recurrent Unit or GRU), number of layers (1, 2, 4), hidden vector size for encoder/decoder (256, 512), and word embedding size for encoder/decoder (256, 512). The experiments are based on 1,018 ROS1 to ROS2 migration pairs. The evaluation results are shown in Figure 85. The hyper-parameter configurations corresponding

to the sequence numbers in Figure 85 are shown in Table 24. The 7th configuration (highlighted in Table 24) has the best performance.

Table 24: Hyper-parameter configurations evaluated in experiments

Sequence	Cell	Layers	Hidden State	Word Embedding
1	GRU	1	256	256
2	GRU	2	256	256
3	GRU	4	256	256
4	LSTM	1	256	256
5	LSTM	2	256	256
6	LSTM	4	256	256
7	<i>GRU</i>	<i>1</i>	<i>512</i>	<i>512</i>
8	GRU	2	512	512
9	GRU	4	512	512
10	GRU	1	128	128
11	GRU	2	128	128
12	GRU	4	128	128

It can be observed that the 7th configuration achieved the best migration performance. This configuration uses GRU, 1 layer, and sets the hidden vector size and word embedding size all equal to 512. With this setting, we can achieve the best performance – 68.2% correct migration on average. Such results signify that, by using the Sequence-to-Sequence model with the correct hyper-parameter, we can achieve a very promising ROS1 to ROS2 migration performance.

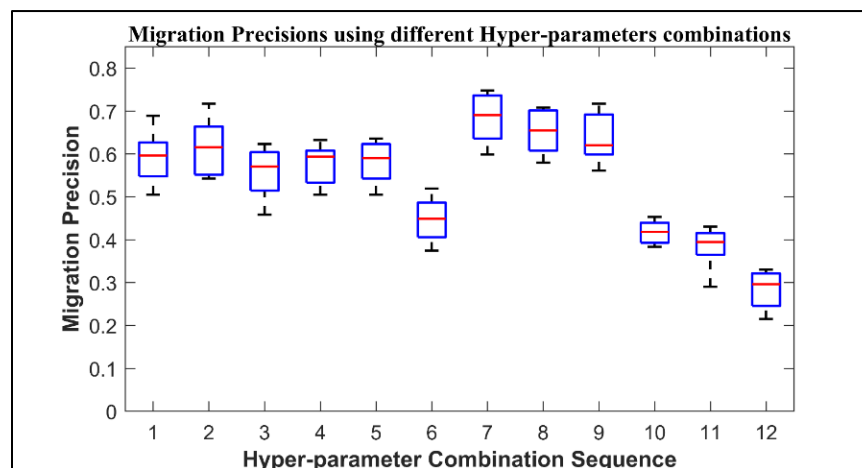


Figure 85: Evaluations of hyper-parameter tuning of the Sequence-to-Sequence model

4.5.2.3.2 External Evaluation by the Voice of the Offense Team (SwRI/GVSC)

GVSC (Ground Vehicle Systems Center) engineers performed evaluation of our ROS code conversion tool. As a basis of comparison, GVSC reported that a manual conversion of a ROS1 package to ROS2 requires 1 FTE 6 weeks. GVSC uses over 200 ROS packages. The evaluation centered on converting ROS1 (Kinetic) packages to ROS2 (Dashing) using unit-level testing and qualitative review for evaluation. The evaluation packages were not disclosed to performers and consisted of the vehicle management system (VMS) code and its dependencies.

The following source code level aspects of ROS2 conversion were evaluated:

- Message/service/action paths, namespaces, and file extensions
- Minor changes to publisher/subscriber syntax; e.g., “ros” and “rclcpp” namespace
- Thread spinning
- Use of std libraries instead of boost
- New/heavily modified interfaces
 - Ros::Time
 - Node creation
 - Logging
 - Parameters

We received training data consisting of:

- ROS packages of the RealTime Kinematic (RTK) “motion executor” control code and associated dependencies.

The evaluation criteria was as follows:

- Ease of conversion (how easy was it to follow the instructions for conversion to generate the ROS2 code?)
- How close the solution came to produce adapted code that can be compiled and run in the designated Ubuntu 18.04 docker container?
 - GVSC noted that building running adapted code was ambitious, and no performer was able to achieve that. This metric qualitatively evaluated the quality of the cobversion.
- Is the migrated code human readable? How easy would it be for a human to improve upon the migration or debug the resulting code?
- Does the solution correctly automate rote, API-like changes?
- How well does the solution perform on the training data?
- How well does the solution identify impacted code even when the resulting migration is incorrect?
- Does the solution offer a user interface with useful features?

The quantitative evaluation result is summarized in Table 25 .

Table 25: ROS conversion results

Total Score	15/50
Ease of Conversion	8/10
Buildable source code	4/10
Human Readability	3/10

The qualitative performance summary is as follows:

- Migration results built with “colcon build” command: 4/10
 - Converted test files have many invalid, “garbled” lines
 - Conversion left syntax errors in the adapted code, which required manual line-by-line correction
 - Credit awarded for the following:
 - Partially handled ROS 2 convention of using std namespace
 - Correctly converted ros.h to rclcpp.hpp.
 - Partially handled publisher API changes but not message types.
 - Partially handled rclcpp::spin(), but with the incorrect syntax.
- Human Readability: 3/10
 - Test files have numerous errors that are not valid C++; e.g., invalid extra spaces between namespaces.
 - Converted training files have similar, though fewer, syntax errors.

We fully anticipated that, as the first attempt to automatically translate ROS1 code to ROS2 using machine learning with limited training data, it will be hard to produce adapted code that will compile and run. Therefore, our *human in the loop* solution offered an interactive solution integrated with a visual editor, where the conversion tool does the legwork to identify which lines need to change and recommend ranked choices for how to change the line such that a human developer can look at the choices, and select the translation that is right, or close and apply whatever edits are necessary to make it right. The evaluation team noted that the interactive mode for collaborative human machine conversion that IMMoRTALS provided is intriguing. They also commented that report generation and conversion metrics could be useful since it informs the user where to start debugging/fixing the source.

5.0 CONCLUSIONS

The research performed by the IMMoRTALS team over the four year period of performance achieved various levels of success in different areas of investigation, delivering significant advances in the following:

- Combining program analysis with semantic annotations,
- Application of relational learning to evolve code that uses SQL queries with respect to schema changes,
- Application of machine learning in program translation,
- Type-based resource modeling and provisioning using efficient variational analysis, and
- Source and bytecode level program rewriting.

The project established the feasibility of a new knowledge-driven toolchain-based approach to program building and maintenance, where software metadata and domain knowledge are used to customize the application code prior to deployment, addressing the deployment specific requirements and ecosystem constraints. While the end to end operation of a proof of concept DAS was demonstrated on a small number of use cases (challenge problems), a lot more research, and thorough test and evaluation are needed for its long terms viability and success. In particular, the demonstrated approach assumes the existence of (a) annotated software that can be used for adding to and extending the current application with new or different features and capabilities, and (b) a well-defined architecture with identified choice points for the software to be adaptable by the automated tools. These assumptions are not commonly valid in the general case, although as AI- and ML-based techniques intrude into every aspect of our lives, software engineering will also see a rise in data and knowledge driven approaches for program understanding, software construction and maintenance. The tools and practices will invariable lead to code artifacts annotated with additional machine analyzable metadata and knowledge, as well as software systems with machine identifiable scaffolding. Therefore, we believe that in the near future these assumptions will become more justifiable.

IMMoRTALS made substantial contribution in the area of knowledge representation to link software artifacts with software metadata; program analysis that combines static and dynamic analysis with semantic annotation to identify dependencies, resource usage, and intra-method, inter-method, and inter-process dataflows that can be checked against constraints over semantics of the data being handled by the dataflows; and automated program rewriting techniques- both at the source and bytecode level based prescribed patterns (as in PQL), and using proxy and API weaving. Work in IMMoRTALS made significant advances in relational learning and its use in learning the query when the database to be queried has a new schema; and variational analysis and the use of a type-system formalism to manage resource provisioning.

In addition to end to end demonstration of the IMMoRTALS DAS performing code adaptation in response to various evolutionary pressures, IMMoRTALS also developed streamlined tools that package specific analysis and adaptation techniques for a given specific type of software and a given specific type of evolutionary pressure. Most notable achievements in this category are the code rewriting that uses prescribed patterns to identify code location to modify as well as how to rewrite the code in that location, and a ML-based tool to translate (both offline and interactive)

ROS1 packages to ROS2. The latter tool can be retrained to translate between different versions of ROS, or other code migration problems. We have begun to apply the same approach in another context, for example, to translate code using one version of JSON library (See Appendix B).

In summary, IMMoRTALS was a good initial step that highlights the need for more research, and at the same time establishes that the meta-data and knowledge driven approaches to software engineering, specifically handling software evolution and maintenance in response to ecosystem changes is a viable research direction. The next section outlines a few recommended next steps.

6.0 RECOMMENDATIONS

Based on our experience in the IMMoRTALS project, we make the following recommendations for short and long term next step research. For the short term,

- *Integration with synergistic DARPA research results/ongoing activities such as MUSE and IDAS:* MUSE developed capabilities to mine big corpus of programs to derive salient features about code—propositions and properties of code signifying their specification, relationships, vulnerabilities among others. Subjecting available code corpus to MUSE is likely to provide valuable metadata that toolchains like IMMoRTALS can make use. IDAS is aiming to develop mechanisms to specify intent and using it to generate and evolve code. IDAS toolchain can potentially be invoked from IMMoRTALS workflow to facilitate code repair. More work is needed to develop the *integration glue* based on the synergies of these diverse but related R&D efforts.
- *Thorough evaluation of actual cost savings of autonomic adaptation:* Understanding of the actual cost savings, say in terms of man hours saved is necessary to validate the effectiveness of BRASS tools like IMMoRTALS. We note that it is not easy or simple to perform such evaluation objectively, and such an effort may in fact require a carefully crafted human subject research and novel metrics for software qualities.
- *Determining what is the best way to deal with the adapted code:* Are these disposable? What needs to happen if we are to beneficially persist the machine adapted code that could at some point in the future will have to be read and edited by a human developer?
- *Focus on how to take advantage of the positive aspects of the change/new deployment environment:* The evolutionary pressure is not necessarily just different/orthogonal requirements or resource restrictive, but could potentially be beneficial and improve a specified software metric (e.g., performance). Our own work so far primarily focused on evolutionary pressure that is either orthogonal or restrictive.
- *Streamlined usable packaging of adaptation technology:* Instead of broad-based big picture resource adaptive applications, focus on combining selected core technologies in streamlined packages (e.g., Gradle plugins) that are self-complete and work on a turnkey basis to solve specific code migration and evolutionary pressure for technology adoption and community penetration.

In terms of longer term research, we argue that availability of semantics and intent (manually added or programmatically derived) that can be combined with additional software metadata, the kinds of which have been used and demonstrated in IMMoRTALS including structural information, metadata about packaging and program building, compilation and testing, data and control flow information etc. — opens up a wide area of research for data and knowledge driven approaches to engineering software and systems. Applications of data and knowledge driven approaches can potentially lead to tools and techniques, among others, for:

- Automatically improving the program written by a novice to a program written by experts,
- Automatically generating complex programs from simpler and smaller code components that meet specified intent or specification through use of solvers,

- Generating well-defined and accurate API and resource access profiles of programs that can be checked/enforced at runtime, and
- Instrumenting generated code injected with code-fragments for self-checking, intent verification or specification conformance at strategic locations.

Given our dependence on software, and the current technology landscape where US and DoD developed software are increasingly deployed in an untrusted hardware and software ecosystem that is rapidly evolving- it is critical that more research is undertaken in this important area.

7.0 REFERENCES

- [1] B. Michael, S. Bugiel, and Erik Derr. “Reliable third-party library detection in android and its security applications.” In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 356-367. ACM, 2016.
- [2] D. Yaniv and E. Yahav. “Tracelet-based code search in executables.” In *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 349-360. ACM, 2014.
- [3] B. Gogul, R. Gruian, T. Reps, and T. Teitelbaum. “CodeSurfer/x86—A platform for analyzing x86 executables.” In *International Conference on Compiler Construction*, pp. 250-254. Springer, Berlin, Heidelberg, 2005.
- [4] F. Jean-Rémy, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. “Fine-grained and accurate source code differencing.” In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 313-324. ACM, 2014.
- [5] S. Deerwester, S. T. Dumais, G.W. Furnas, T. K. Landauer, and R. Harshman. “Indexing by latent semantic analysis.” *Journal of the American society for information science* 41, no. 6 (1990): 391-407.
- [6] H. Kagdi, M. Gethers, and D. Poshyvanyk. “Integrating conceptual and logical couplings for change impact analysis in software.” *Empirical Software Engineering* 18, no. 5 (2013): 933-969.
- [7] D. Bahdanau, K. Cho, and Y. Bengio. “Neural machine translation by jointly learning to align and translate.” *arXiv preprint arXiv: 1409.0473* (2014).
- [8] A. See, P. J. Liu, and C. D. Manning. “Get To The Point: Summarization with Pointer-Generator Networks.” In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073-1083. 2017.
- [9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. “Attention is all you need.” In *Advances in neural information processing systems*, pp. 5998-6008. 2017.
- [10] H. Wang, Y. Guo, Z. Ma, and X. Chen. “WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection.” In *Proceedings of the International Symposium on Software Testing and Analysis*, 71–82. ISSTA 2015. New York, NY, USA: ACM. (2015)
- [11] S. Paul and A. Prakash. “A Framework for Source Code Search Using Program Patterns.” *IEEE Transactions on Software Engineering* 20 (6): 463–75. 1994
- [12] J. Picado, A. Termehchy, A. Fern, and P. Ataei. “Schema Independent Relational Learning.” *ArXiv:1508.03846 [Cs]*, November. <http://arxiv.org/abs/1508.03846>. 2017
- [13] https://www.mitre.org/sites/default/files/pdf/09_4937.pdf
- [14] I. Horrocks, P. F. Patel-Schneider, F. Van Harmelen. “From SHIQ and RDF to OWL: the making of a Web Ontology Language”. In *Web Semantics: Science, Services and Agents on the World Wide Web*
- [15] I. Horrocks and U. Sattler. “Ontology Reasoning in the SHOQ(D) Description Logic.” In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.
- [16] “RDF,” RDF — Semantic Web Standards. [Online]. Available: <https://www.w3.org/RDF/>. (last accessed: Dec, 2019).

- [17] SPARQL Query Language for RDF. [Online]. Available: <https://www.w3.org/TR/rdf-sparql-query/>. [Accessed: 06-Dec-2019].
- [18] T. Berglund and M. McCullough. “Building and Testing with Gradle.” O’REILY, 2011
- [19] “Apache Jena Fuseki,” Apache Jena — Apache Jena Fuseki. [Online]. Available: <https://jena.apache.org/documentation/fuseki2/>. (last accessed: 05-Dec-2019).
- [20] L. K. Yan, H. Yin. “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.” In *Proceedings of the 21st Usenix Security Symposium*, Bellvue, WA, August 2012.
- [21] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. “Semantics-Aware Android Malware Classification using Weighted Contextual API Dependency Graphs”, In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS’14)*, Scottsdale, AZ, November 2014.
- [22] L. Lu, Z. Li, Z. Wu, W. Lee, G. Jiang. “CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities.” In *Proceedings of Computer and Communication Security (CCS)*, pp 229-240, Raliegh, NC, October 2012.
- [23] M. Erwig and E. Walkingshaw. “The Choice Calculus: A Representation for Software Variation.” *ACM Trans. On Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.
- [24] M. Erwig and E. Walkingshaw. “Variation Programming with the Choice Calculus.” In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 55–99, 2012.
- [25] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. “Variational Data Structures: Exploring Trade-Offs in Computing with Variability.” In *ACM SIGPLAN Symposium on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.
- [26] S. Chen, M. Erwig, and E. Walkingshaw. “A Calculus for Variational Programming.” In *Proceedings of the European Conf. on Object-Oriented Programming (ECOOP)*, pages 6:1–6:26, 2016.
- [27] A. Christi, A. Groce and R. Gopinath, “Resource Adaptation via Test-Based Software Minimization,” 2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Tucson, AZ, 2017, pp. 61-70.
- [28] A. Christi and A. Groce, “Target Selection for Test-Based Resource Adaptation,” 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), Lisbon, 2018, pp. 458-469.
- [29] A. Christi, A. Groce and R. Gopinath, “Evaluating Fault Localization for Resource Adaptation via Test-Based Software Modification,” 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), Sofia, Bulgaria, 2019, pp. 26-33.
- [30] J. A. Jones and M. J. Harrold, “Empirical evaluation of the Tarantula automatic fault-localization technique,” in *Automated Software Engineering*, 2005, pp. 273–282.
- [31] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.
- [32] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

- [33] G. Misherghi and Z. Su, “HDD: Hierarchical delta debugging,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06, 2006, pp. 142–151.
- [34] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, “Cause reduction: Delta-debugging, even without bugs,” *Journal of Software Testing, Verification, and Reliability*, accepted for publication.
- [35] —, “Cause reduction for quick testing,” in *IEEE International Conference on Software Testing, Verification and Validation*, 2014, pp. 243–252.
- [36] E. Wong, T. Wei, Y. Qi and L. Zhao, “A Crosstab-based Statistical Method for Effective Fault Localization,” *2008 1st International Conference on Software Testing, Verification, and Validation*, Lillehammer, 2008, pp. 42-51.
- [37] Measurement Description Language (MDL), IRIF 106, Chapter 23, available from <http://www.irig106.org/docs/106-17/chapter23.pdf> (last accessed Feb, 2020).
- [38] DROOLs webpage, <https://www.drools.org/> (last accessed Feb, 2020)
- [39] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. *Variational Data Structures: Exploring Trade-Offs in Computing with Variability*. In *ACM SIGPLAN Symp. On New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014
- [40] A. Grasley. “Imperative Programming with Variational Effects.” MS thesis. Oregon State University, 2018.
- [41] M. Meng, J. Meinicke, CP Wong, E. Walkingshaw, and C. Kästner. “A Choice of Variational Stacks: Exploring Variational Data Structures.” In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2017.
- [42] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. “Soot — a Java bytecode optimization framework.” In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON ’99)*, page 13. IBM Press, 1999
- [43] E. Bruneton. “ASM 3.0, a Java bytecode Engineering Library”, <http://download.forge.objectweb.org/asm/asm-guide.pdf>, 2007
- [44] S. Chen, M. Erwig, and E. Walkingshaw. “Extending Type Inference to Variational Programs.” *ACM Trans. On Programming Languages and Systems (TOPLAS)*, 36(1):1:1–1:54, 2014.
- [45] S. Chen, M. Erwig, and E. Walkingshaw. “An Error-Tolerant Type System for Variational Lambda Calculus.” In *Proceedings of ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 29–40, 2012.
- [46] S. Chen and M. Erwig. “Early Detection of Errors in C++ Templates.” In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pp. 133–144, 2014.
- [47] S. Chen and M. Erwig. “Counter-Factual Typing for Debugging Type Errors.” In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 583–594, 2014.
- [48] S. Chen and M. Erwig. “Principle Type Inference for GADTs.” In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 416–428, 2016.
- [49] J. P. Campora III, S. Chen, M. Erwig, and E. Walkingshaw. “Migrating Gradual Types.” In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, volume 2, pages 15:1—15:29, 2018.

- [50] J.P. Campora III, S. Chen, and E. Walkingshaw. "Casts and Costs: Harmonizing Safety and Performance in Gradual Typing." In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 2, pages 98:1—98:30, 2018.
- [51] Financial products Markup Language (FpML) webpage at <https://www.fpml.org/> (last accessed Feb, 2020)
- [52] FpML v5.11 and v5.10 documentation, available at https://www.fpml.org/the_standard/current/ (last accessed Feb 2020)
- [53] SOAP, Simple Object Access Protocol, Documentation available at <https://www.w3.org/TR/soap/> (last accessed Feb, 2020)
- [54] M.L.Collard, J.I. Maletic, B. Robinson. "A Lightweight Transformational Approach to Support Large Scale Adaptive Changes", In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, Timisoara, Romania, Sep 12-18 2010.

APPENDIX A - DELIVERABLES AND PUBLICATIONS

- Deliverables
 - Tech reports
 - CP descriptions (once in each phase)
 - API documentation and user guides (once in each phase)
 - Software
 - Code drop containing IMMORTALS technology and solution to CPs (once in each Phase)
 - In addition the following software is available as open source
 - The variational SMT solver is available at: <https://github.com/doyougnu/Vsat>
 - The variational execution monads are added as a package to Haskell's package repository (<https://hackage.haskell.org/>)
 - The resource DSL is available at: <https://github.com/lambda-land/Resource-DSL-Deploy>
 - The variational database management system is available at: <https://github.com/lambda-land/VDBMS>
 - Castor is available at <https://github.com/OSU-IDEA-Lab/Castor>
 - AutoMode is available at <https://github.com/OSU-IDEA-Lab/AutoMode>
 - Presentation
 - For Kickoff (Dec 2015), and each PI meeting (Jul 2016, Mar 2017, Nov 2017, Jul 2018, Mar 2019, Nov 2019)
 - For site visits (Apr 2016, Feb 2017)
 - Status reports
 - Monthly single slide summary
 - QTRs in TFIMS (every Quarter CY)
- Publications
 - J. Staples, C. Endicott, L. Krause, P. Pal, P. Samouelian, R. Schantz, A. Wellman. "A semi-autonomic bytecode repair framework", in *IEEE Software*, vol. 36, no. 2, pp. 97-102, 2019.
 - Y. Duan, X. Li, J. Wang, and H. Yin. "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing", to appear in the *Network and Distributed System Security Symposium (NDSS)*, February 2020
 - Y. Duan, L. Gao, J. Hu, and H. Yin. "Automatic Generation of Non-intrusive Updates for Third-Party Libraries in Android Applications", in the *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, September 2019
 - Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang. "Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation", in the *Network and Distributed System Security Symposium (NDSS)*, February 2018

- A. Christi, A. Groce, A. Wellman. “Building Resource Adaptations via Test-Based Software Minimization: Application, Challenges, and Opportunities”, in *IEEE 30th International Symposium on Software Reliability Engineering (ISSRE-Industry Track)* 2019, Berlin, Germany.
- A. Christi, A. Groce, R. Gopinath. “Evaluating Fault Localization for Resource Adaptation via Test-Based Software Modification”, in *IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)* 2019, Sofia, Bulgaria.
- A. Christi and A. Groce. “Target Selection for Test-Based Resource Adaptation”, in *IEEE International Conference on Software Quality, Reliability and Security (QRS)* 2018, Lisbon, Portugal.
- A. Christi, A. Groce, and R. Gopinath. “Resource Adaptation via Test-Based Software Minimization”, in *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)* 2017, Tuscon, AZ.
- J. Picado, A. Termehchy, A. Fern, and P. Ataei. “Logical Scalability and Efficiency of Relational Learning Algorithms”, in *the VLDB Journal (VLDBJ)*, 2018
- J. Picado, A. Termehchy, and S. Pathak. “Learning Efficiently over Heterogeneous Databases”, in *proceedings of the VLDB Endowment (PVLDB)*, 2018
- J. Picado, A. Termehchy, and S. Pathak. “Learning Efficiently Over Heterogeneous Databases: Sampling and Constraints to the Rescue”, in *proceedings of the Second Workshop on Data Management for End-to-End Machine Learning (DEEM)*, 2018
- J. Picado, S. Pathak, A. Termehchy and A. Fern. “AutoMode: Relational Learning with Less Black Magic”, in *proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2018
- J. Picado, A. Termehchy, A. Fern, and P. Ataei. “Schema Independent Relational Learning”, in *proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017
- J. Picado, P. Ataei, A. Termehchy, and A. Fern. “Schema Independent and Scalable Relational Learning by Castor”, in *proceedings of the VLDB Endowment (PVLDB)*, 2016
- Y. Chodpathumwan, J. Picado, A. Termehchy, A. Fern and Y. Sun. “Toward Representation Independent Analytics over Structured Data”, in *Data Wrangling Automation workshop in conjunction with IEEE ICDM*, December 2016
- S. Hubbard and E. Walkingshaw. “Formula Choice Calculus”, in *International Workshop on Feature-Oriented Software Development (FOSD’16)*, 2016
- M. Meng, J. Meinicke, C-P. Wong, E. Walkingshaw, and C. Kästner, “A Choice of Variational Stacks: Exploring Variational Data Structures”, in *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS’17)*, February 2017
- R. Gopinath, and E. Walkingshaw. “How Good Are Your Types? Using Mutation Analysis to Evaluate the Effectiveness of Type Annotations”, in *International Workshop on Mutation Analysis (Mutation’17)*, March 2017

- P. Ataei, A. Termehchy, and E. Walkingshaw. “Variational Databases”, in International Symposium on Database Programming Languages (DBLP’17), 2017
- J.P. Campora, S. Chen, M. Erwig, and E. Walkingshaw. “Migrating Gradual Types”, in *Principles of Programming Languages (POPL’18)*, 2018
- J. P. Campora, S. Chen, and E. Walkingshaw. “Casts and Costs: Harmonizing Safety and Performance in Gradual Typing”, in *International Conference on Functional Programming (ICFP’18)*, 2018
- S. Adelsberger, A. Setzer, and E. Walkingshaw. “Developing GUI Applications in a Verified Setting”, in *Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA’18)*, 2018
- S. Adelsberger, A. Setzer, and E. Walkingshaw. “Declarative GUIs: Simple, Consistent, and Verified”, in *International Symposium on Principles and Practice of Declarative Programming (PPDP’18)*, 2018
- P. Ataei, A. Termehchy, and E. Walkingshaw. “Managing Structurally Heterogeneous Databases in Software Product Lines”, in *the VLDB Workshop on Polystores and Other Systems for Heterogeneous Data (Poly’18)*, 2018
- T. Thüm, L. Teixeira, K. Schmid, E. Walkingshaw, M. Mukelabai, M. Varshosaz, G. Botterweck, I. Schaefer, and T. Kehrer. “Toward Efficient Analysis of Variation in Time and Space”, in *International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution’19)*, 2019

APPENDIX B- ADDITIONAL EXPLORATION ML-BASED CODE TRANSLATION

Toward the end of the base performance period, the IMMoRTALS team was granted a number of no cost extensions for various reasons. The BBN team utilized the available time to perform additional exploration of ML-based code translation. This exploration was performed in the context of JSON (Java Simple Object Notation) library usage, specifically, migrating code that used an earlier version of a JSON library to a newer version of the same library. Our goal was to apply machine translation to migration of such code, and also to improve our understanding of various issues identified during evaluation of our work on automated ROS code migration. The scope of the exploration was limited by the amount of resources

JSON is a data representation format for storing Java object on file as well as sending Java objects over the wire. We investigated migrating code impacted by both structural and behavioral API changes in the popular Jackson-Core library. Jackson-Core is a widely used library for working with JSON. At the time of this report, Maven Central reported 6,953 artifacts which depend on Jackson-Core. The project forms the core of other JSON libraries (e.g., software artifacts produced by Redhat, Apache as well as other Jackson-core based JSON libraries like Jackson Databind). Importantly, the library provides a long-running history of commits essential for studying the impact of evolving APIs, including 102 releases and 2,109 Git commits since December 2011.

Specifically, we started with a subset of the APIs that are deprecated in the most recent version of Jackson core (version 2.11.0.rc1)¹³, and evaluated how well a translator trained on samples code using the API before and after they were deprecated migrates novel samples of deprecated API usage to valid Jackson core usage.

Our prior work on ROS-1 to ROS-2 migration, we applied a sequence-to-sequence neural network model borrowed from machine translation research on natural languages. The training data consisted of C++ ROS-1 code and its corresponding ROS-2 counterpart; i.e., the approach required a corpus of migrated ROS-1 code. This approach encountered a number of significant challenges, in this limited exploration we focused on two of them:

- Training data (lack thereof)
- Input representation

API Changes Considered

We explored the impact on Jackson-Core clients of the following API changes. The *Migration* column below is in the format *from-api* -> *to-api* (i.e., clients were instructed as of the version in the *Deprecated* column to migrate any code using the signature *from-api* to the *to-api* signature).

¹³ Published in <https://www.javadoc.io/doc/com.fasterxml.jackson.core/jackson-core/latest/deprecated-list.html>

Table B-1: Studied code migrations

Deprecated	Migration (<i>from-api</i> -> <i>to-api</i>)
2.2	JsonFactory.createJsonParser(byte[]) -> JsonFactory.createParser(byte[])
2.1	JsonFactory.createJsonParser(File) -> JsonFactory.createParser(File)
2.1	JsonFactory.createJsonParser(InputStream) -> JsonFactory.createParser(InputStream)
2.1	JsonFactory.createJsonParser(Reader) -> JsonFactory.createParser(Reader)
2.1	JsonFactory.createJsonParser(String) -> JsonFactory.createParser(String)
2.1	JsonFactory.createJsonParser(URL) -> JsonFactory.createParser(URL)
2.8	createFlattened(JsonParser, JsonParser) -> createFlattened(boolean, JsonParser, JsonParser)
2.8	_reportInvalidEOF(String) -> _reportInvalidEOF(String, JsonToken)
2.10	JsonFactory.configure(JsonFactory.Feature, boolean) -> TSFBuilder.configure(JsonFactory.Feature, boolean)

We used the same sequence-to-sequence based technique to train a neural network model for these JSON API changes that we used for ROS migration. In this exploration, we used the trained model to predict the new API usage for new or set aside samples and manually looked at top 3 predictions to determine whether they are valid translation or not. As with ROS migration, in this limited exploration we also observed that:

- Newly evolved APIs lack adequate training samples. For example, we were unable to find any examples for the 2.10 deprecation in Table B-1 on Github, Program Creek or Codota. Our work on the ROS migration faced similar limitations. In other cases, there are many training samples on the use of the older API and only a few samples for the newer API. This imbalance is to be expected as a newer API gains wider use and it would be advantageous to develop a technique that can make use of available samples as few as they may be. The use of deprecation (versus a hard API change) also makes it difficult to find training samples that are properly paired; i.e., before-after versions of the same project that contain the API migration.
- Even when suitable examples are available, significant engineering effort is required to produce the training pairs. Code projects must be selected based on multiple criteria, including a dependency on the API library that has evolved as well as the existence of a subsequent *minor* upgrade that includes a dependency on the new version of the API. Establishing this correspondence is very difficult since public code repositories like Github and Maven currently expose a limited query interface. It is much easier in many cases to simply find available samples of each API version, especially when leveraging existing sites such as Program Creek (<https://www.programcreek.com/java-api-examples/?action=search>) and Codota (<https://www.codota.com/code>).

In the remainder of this Section we will discuss the issues of training data and input representation, and our approach to mitigate them, and then describe our evaluation results.

The Issue of Training Data

The lack of training data was likely at the core of many of the issues identified during evaluation, including invalid or garbled lines and syntax errors in the predicted code. Consider the following JSON API call in Figure B-1:

```
static JsonParserSequence createFlattened(JsonParser first, JsonParser second)
```

Method that will construct a parser (possibly a sequence) that contains all given sub-parsers

Figure B-1: JSON sample API

The method accepts two instances of `JsonParser`. A client may supply these arguments at a call site in the form of variable references or as results of function calls; e.g. as shown in Figure B-2:

```
JsonParserSequence localJsonParserSequence =  
    JsonParserSequence.createFlattened(localTokenBuffer.  
        asParser(paramJsonParser), paramJsonParser);
```

Figure B-2: Sample JSON code for createFlattened API

This results in a vast space of options that are tied to the context and semantics of the code fragment where the call is situated (to see that, consider how many ways it is possible to get a `Boolean` and two `JsonParser` objects, and each program will have a different set of sources and methods to get them) that are very sparsely filled with the training samples we have. Our approach as applied to ROS and JSON both show that while machine translation methods are able to repair simple function or method name changes, they are not able to produce meaningful predictions in many cases on argument values. Such predictions reveal overfitting on the specifics of the training data (as shown in Figure above) that are most likely not relevant for novel cases

Mitigation approach taken in this study- boosting training data: We decided to investigate the benefits and issues in an alternate approach to building training data by creating a corpus consisting of samples from each API version versus finding projects that made use of each API version across two commits. To produce the samples, we queried sites like Program Creek and Codota using a simple search seed that consisted of the method and library name. This effectively filtered similarly named methods from other libraries and highlighted the relevant lines of code in the results. Only the method invocation was selected. As with our approach on the ROS migration, this limited our approach to migrating simple statements.

Since our training samples no longer consist of old and new API versions of the same code, we can take advantage of this situation further by creating the Cartesian product between the samples we find to further boost our training data. This also addresses the imbalance discussed earlier that exists between old and new samples of an API. If there are no samples available, we can synthesize one based on any of the old API samples. While this boosted our training data significantly, it resulted in pairs of samples whose only relation was that they represented the respective API usages correctly. To mitigate this, we filtered the code pairs using the levenshtein distance function with a threshold of 35 under the premise that similar code samples would yield more relevant predictions on the same token sequence in novel code.

This approach resulted in 172 records which we split as follows across training, evaluation and test (80-10-10). The test set cannot be evaluated by simply comparing between the old and new API samples since these are no longer correlated. Hence, a manual qualitative review was applied. For each test sample and prediction, we determined whether the predicted sample showed a correct sample of the new API (which was expected) and whether the prediction would help an engineer repair or migrate the target code by showing how the new API relates to the old and what the nature of the change was. The training data highlighted three types of refactoring:

1. Method name changes
2. Method signature changes
3. Methods being refactored from one class to another.

The Issue of Input Encoding for Learning the Model

Tokenization: Our machine translation efforts on ROS and JSON both utilized the OpenNMT 2.8 library (the ROS work made use of a version built on PyTorch whereas for the JSON study we used the TensorFlow version). OpenNMT is built for machine translation tasks applied on natural languages. Sequence-to-sequence neural network models in this domain ultimately begin by building a vocabulary where each ‘word’ is assigned a one-hot encoding that can then be fed one at a time into the neural network. Tokenization algorithms are key to producing a meaningful vocabulary. The available tokenization strategies offered by OpenNMT are designed for natural languages. They place emphasis on spaces and natural language punctuation marks. A tokenization strategy for code, on the other hand, must be aware of the semantics of the target language; e.g., the semantics of parenthesis, dot operators, etc. For example, the text “+_currToken” should result in two tokens in the context of programming languages, but a natural language tokenizer may produce a single token. Aggressive tokenization may yield 3 separate tokens: “+”, “_”, “currToken”, which would also be incorrect in the context of most programming languages.

Matters become more complex with examples like these:

```
1. _reportInvalidEOF("in a value") ;
2. _JsonParser parser =
   jsonFactory.createJsonParser("{ \"name\": \"John\", \"grade\": \"D\" } ") ;
3. JsonParser jp = f.createParser (new File ("user.json" ) ) ;
```

In example 1 above, a tokenization based on spaces would create separate tokens for the quoted value (“in a value”). In the context of machine translation using sequence-to-sequence models, this is not likely a meaningful division and the entire text should be treated as a single token. This is similar to usage based consolidation that occurs in text mining; e.g., the meaning of ‘dead end’ is similar in both cases below:

- This road leads to a dead end.
- He was stuck at a dead-end job.

Tokenization strategies used in natural languages may consolidate these two to a single representation. Special dictionaries and detailed rules are often used to find the right level of tokenization in text mining. In many cases, practitioners must try different strategies to discover an optimal approach for each case. A similar effort is likely required for machine translation approaches on code artifacts. Example 2, taken from the JSON corpus, shows how complex tokenization can be. For the JSON study, the entire quoted value, along with the backslash escapes, was represented as one token. Finally, in example 3 we see that the semantics of symbols is very context sensitive. The dot or period in “user.json” should not be used as a token delimiter whereas the dot in “f.createParser” clearly separates an object receiver from a method invocation and should be used to split the value into two tokens. Spaces may be used to some extent to ensure values are tokenized correctly, but this may lead to invalid code predictions. One of the evaluation comments was that there were numerous errors in predicted values, which often produced invalid C++ code that contained invalid extra spaces between namespaces. This was likely the result of tokenization issues.

Mitigation approach taken in this study: An optimal tokenizer should make use of the code’s parse tree and language grammar. For the limited resources of the short JSON study, we did not have time to build a custom tokenizer and fully research this issue. We manually tokenized data as best as possible using forced spaces as a delimiter. The resulting vocabulary of 113 tokens is shown in Figure B-3 (each cell is a token).

<blank>	tb	msg	TestUtils
<s>	jsonFactory	type	copyResourceToTempFile
</s>	File	TokenBuffer	guardianproject_index-v1.json
(	true	source	writer
)	seq	final	QUOTE_NON_NUMERIC_NUMBERS
.	p1	pb	createJsonGenerator
;	p2	snapshotIn	out
,	json	content	_factory
=	jsonInString	first	allowComments
JsonParser	a	second	mapper
“	value	JsonGenerator	getFactory
JsonParserSequence	file	ALLOW_COMMENTS	jf
parser	jsonfactory	jParser	INCLUDE_SOURCE_IN_LOCATION
jp	localJsonParserSequence	jfactory	ALLOW_TRAILING_COMMA
createFlattened	localTokenBuffer	response	ALLOW_UNQUOTED_FIELD_NAMES
createJsonParser	FACTORY	getStream	INTERN_FIELD_NAMES
createParser	localObject1	{"name":"John","grade":"D"}	tmpJsonFactory
in	seq1	ALLOW_SINGLE_QUOTES	feature
false	p	ALLOW_NON_NUMERIC_NUMBERS	enabled
factory	return	JacksonParser	CANONICALIZE_FIELD_NAMES
_reportInvalidEOF	this	{"k1":"v1"}	
configure	_jsonFactory	JsonFactory	
_currToken	stream	user.array.json	
+ _currToken	Person.json	user.json	
asParser	jsFactory	AUTO_CLOSE_TARGET	
Feature	args	T	
paramJsonParser	[	_readMapAndClose	
new	0	_typeFactory	
AUTO_CLOSE_SOURCE	]	constructType	
f	state	valueType	
TSFBuilder	p3	ByteArrayInputStream	
+			

Figure B-3: Lexicon for JSON samples

Input Representation: One-hot encodings are only the start of the sequence-to-sequence translation model. Tokens internally become represented in an embedding space using vectors. Natural language translation models make use of word2vec and Glove, and OpenNMT supports both of these options.

A key issue that must be considered, however, is the set of components (latent invariant structures that the model should learn from) this embedding space should represent. Consider the following example again:

```
JsonParser parser =
jsonFactory.createJsonParser("{\"name\":\"John\", \"grade\":\"D\"} ");
```

Intuitively, the invariant aspects of the code include type names, method names and operators like ‘=’ and ‘.’. The details of identifier names and string literals, on the other hand, are not likely to be relevant to novel cases. Therefore the embedding space should focus on the latent relationship among the invariants, and between them and the abstracted arguments. If we consider larger blocks of code than a single statement, then relationships, perhaps including semantics of the sequence of

statements will come into the focus of the learning technique. In our current study, we are focusing on individual code lines.

An alternative representation for the above code that represents invariant aspects without case-specific details:

```
$v1 (JsonParser) = $v2(JsonFactory).createJsonParser($v3(String));
```

Abstracting code in this manner may yield improved results with machine translation techniques. If a change is made to the above types or parameter structure, predicted values, which would also be in this same abstract representation, could be used as input to manual, engineering based repair techniques that attempt to fill in the normalized identifier references with suitable values in the overall context of the novel code sample.

One drawback to this approach, however, is that input code samples must be loaded in the context of their overall project so that a symbol solver can effectively resolve all identifier references to type names. This is not trivial and requires a great deal of engineering we did not have time for on the JSON study. An alternative approach is to simplify the representation as follows:

```
$v1 = $v2.createJsonParser($v3);
```

Mitigation approach taken in this study: For the JSON effort, we represented code in two ways: (1) the abstract representation (without type information) shown just above; (2) the raw, as-is code representation from the sample.

Evaluation

In all cases, the model's predictions included the correct API sample. For example, the input sample shown below yielded the following top 3 predictions:

Input:

```
this . $v1 = jsonFactory . createJsonParser ( $v2 ) ;
```

Predictions:

```
jf . createParser ( $v1 )
```

```
JsonParser $v1 = new JsonFactory() . createParser ( $v2 )
```

```
$v1 = factory . createParser ( $v2 ) ;
```

The last prediction is likely the closest to the input value and could potentially be used in an engineering-based solution to repair the input code. Notice the first two predictions are missing a semi-colon (they were likely sourced from samples that contained an inline function call as part of a larger statement and hence did not require a semi-colon). All the responses in the sample were similar qualitatively: producing the correct method name but not all the information necessary to produce a complete and correct repair on the input source. Our conclusion is that this approach

requires much more in-depth research and a more full abstract representation that captures type information.

The next section evaluates the results using the raw inputs. We evaluated 17 test samples and their top 3 predicted values. Consider the following API migration:

```
_reportInvalidEOF(String) ->_reportInvalidEOF(String,JsonToken)
```

This example is particularly interesting since it goes beyond a simple method name change. The top three predicted values are:

```
_reportInvalidEOF ( msg , type ) ;  
_reportInvalidEOF ( " in " + _currToken , _currToken ) ;  
_reportInvalidEOF ( " in " + _currToken , _currToken ) ;
```

The predictions above show the engineer that a method signature change has occurred and, more importantly, a suggestion of how a new parameter (JsonToken) relates to the existing parameter (String). The engineer could potentially fix the target code more quickly using this sample. Note also that this sample highlights a tokenization error introduced by manual pre-processing (the last two samples only differ by a space in +_currToken after the “+” symbol).

An example that involved a simple method name change is shown below:

Input:

```
JsonParser parser =  
jsonFactory.createJsonParser("{ \"name\": \"John\", \"grade\": \"D\" } " );
```

Predictions:

```
JsonParser parser = factory . createParser ( jsonInString ) ;  
JsonParser parser = factory . createParser ( json ) ;  
JsonParser parser = factory . createParser json ) ;
```

The predictions make it easy for the engineer to see how to modify the target code.

An example where a method was migrated to another class is shown below:

Input:

```
jsonFactory . configure ( Feature . AUTO_CLOSE_TARGET , false ) ;
```

Predictions:

```
TSFBuilder.configure (JsonParser.Feature.AUTO_CLOSE_SOURCE, false ) ;  
TSFBuilder.configure (JsonParser.Feature.AUTO_CLOSE_SOURCE, false ) json ) ;  
TSFBuilder.configure (JsonParser.Feature.AUTO_CLOSE_SOURCE, false);TSFBuilder);
```

The first prediction clearly shows that the method is no longer associated with the jsonFactory instance but instead the TSFBuilder. The engineer can also see that the parameter signature has not changed and shows how the two correspond.

These results are summarized below in Table B-2.

Table B-2: PQL results summary

Number of test samples	17
Number of cases where a correct sample of the new API is predict	17 (as expected)
Number of cases where the prediction helps the engineer fix the input code (e.g., by showing how the new API changed or how the new API relates to the original API).	12

APPENDIX C- EXTENDING AND APPLYING PQL IN A TRANSITION CONTEXT

BACKGROUND

After the work on the IMMoRTALS base project was completed, BBN was awarded a small ECP to enhance and mature the Program Query Language-based source code transformation capability, and evaluate the enhanced capability in the context of ATAK and ROGER plugins for potential transition to the TAK/ROGER developer community.

The PQL source code transformation capability was initially developed under the base effort of the IMMoRTALS project (described in 4.5.1.1). PQL realizes a pattern-based source code transformation approach to mitigate the code maintenance impact of *API Evolution*. Modern software development involves use of multiple external libraries. User developed code makes calls to the imported libraries using the published API of the libraries. Being software themselves, libraries also change over time for modernization, bug fixes, feature updates, and security and performance enhancements, and also to keep with the evolution of software and hardware they themselves depend on. Some of these changes in the libraries also change their APIs. The problem of API evolution refers to situations when API changes in a library breaks the software programs that depend on them, and code maintainers need to quickly modify their code so that adoption of the new library version does not make their software obsolete and unusable. Depending on the number of external libraries used and their release cycle, the maintenance task to support the updated library can be significantly large and complex. Automation and tool support is often cited as one of the key elements of the developers' wish list in the context of addressing this issue.

Automatically adapting source code when the semantics of the 3rd party library it depends on is changed involves program understanding, and is extremely hard. Fortunately, according to [Dg]¹⁴, over 80% of API breaking library updates results from semantics-preserving refactoring of the library code. This presents the opportunity - restricting the focus to semantics-preserving API evolution is more tractable, yet it can still help the code maintainers and developers significantly.

PQL aims to reduce the cost of impact analysis and maintenance of software due to refactoring of the libraries and frameworks it depends on. The pattern-based approach taken by PQL provides a DSL for the library developers to express the transformations needed to adapt the client code that depends on the library as patterns, and a DSL compiler for the client code maintainers that can be used to automatically transform their code. Each library refactoring can be captured in a DSL program or in a DSL program and the compiler packaged as a single executable, which the library vendors can include in their release package each time they release a library update.

The Android Team Awareness Kit is a tactical information management platform that is widely used by various civilian and military agencies. It consists of a server (TAK Server) and a cursor-on-target (CoT) map-based client App (ATAK Clients) that runs on Android devices. The ATAK solution is frequently used with the Robust Operational Gateways Enabling Reachback/Reach-forward (ROGER) technology. ROGER provides a Gateway for other back end systems and services that are not inherently compatible with the ATAK framework (e.g., CoT messaging) to be used in tactical information management. Both the ATAK framework and the ROGER Gateway

¹⁴ [1] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," p. 26, 2006.

are designed with extensibility in mind, and realize a plugin architecture. BBN has been the primary developer of the TAK Server and the ROGER Gateway technologies. Since ATAK and ROGER is being used in the tactical user community that operate in a highly dynamic environment and are increasingly called to support new and different missions, BBN receives a steady stream of change requests and requests for new functionality for the TAK Server and the ROGER gateway. In the meantime, a number of organizations has started developing context- and mission-specific functionalities that can be delivered to the tactical users via the ATAK Client in the form of plugins in order to support the demand for new capabilities and usage contexts. It is no surprise that the plugins have dependencies on TAK Server and ROGER, and when new versions of TAK Server or the ROGER gateway are released, some of the existing plugins may break.

We saw a transition opportunity to apply the PQL technology we developed under the IMMoRTALS base effort to address the API evolution problem the ATAK and ROGER ecosystem is facing, which led to the ECP at the end of the base effort. The goal was to make PQL more usable by ROGER and TAK Server developers who are not IMMoRTALS or PQL experts, so that they can release a transformation package alongside the new versions of TAK Server and ROGER, that will (a) scan through the plugin source code and identify code points that are impacted by the semantic-preserving API changes in the new release, (b) make the transformation by preserving the programming language (Java in this case) syntax as best as possible (so that the resulting code can compile), and (c) generate a summary report the identified locations and changes made.

OBJECTIVE

The main technical goals of the ECP are to

- Further develop PQL's core code pattern matching and refactoring capabilities to make sure we support the basic semantic-preserving API evolution cases, and more specifically, the kind of API changes the TAK Server and ROGER developers faced in the context of 3rd party plugins.
- Expand and simplify the PQL language (a DSL) to include additional concepts and provide an easy to use syntax so that ROGER and TAK Server developers (primarily Java developers) that are not part of the IMMoRTALS team or PQL experts can use it to express the transformation needed to make the plugins compatible with the new TAK Server and ROGER versions.
- Enhance the tool chain so that it is easier for TAK Server and ROGER developers to develop and release the transformation package, and also to make it easier for the plugin developers to apply the transformation package on their plugin source.
- Apply/evaluate the extended capability to ROGER/ATAK use cases.

RELATED WORK AND POTENTIAL ALTERNATIVE APPROACHES

In the IMMoRTALS base effort, we researched and developed three approaches to evolve software to accommodate external pressures to change.

Protocol/Usage specification: The idea here is that library developers annotate (using an agreed upon ontology) library code with the details of how clients should use the functionality exported by the library, for example, this function initializes the instance of a library class, this function performs this task i.e., usage pattern or protocol to use the classes and functions the library exposes (see Figure C-1). Given this, swapping one implementation with another e.g., a different library implementing the same functionality or a different version of the same library can be achieved by programmatically replacing the code points using the current function with code following the usage pattern of the new implementation or new version. We built the tooling to realize this approach, and used the case of switching an Android Location Provider component in the ATAK Lite client with a Secure Location Provider to demonstrate and evaluate the tool. While this approach is promising, it presents a number of significant technical challenges in the context of API evolution and wider acceptance:

```
public class LocationProviderAndroidGpsBuiltin {  
  
    @FunctionalAspectAnnotation(aspect =  
        InitializeAspect.class)  
    public void initialize(AndroidContext ctx) {  
        //set-up provider's qualities of service ...  
    }  
  
    @FunctionalAspectAnnotation(aspect =  
        GetCurrentLocationAspect.class)  
    public synchronized Coordinates getLastKnownLocation() {  
        //return current location ...  
    }  
  
    @FunctionalAspectAnnotation(aspect =  
        CleanupAspect.class)  
    public void close() {  
        //clean-up resources ...  
    }  
}
```

Figure C-1: Usage pattern annotation of a Location Provider implementation

- The presence of usage-pattern metadata does not provide information on *how* that usage-pattern should be mapped to a call site in the client code. The client code also needs some form of annotations identifying code points invoking imported functions, and also the arguments used in such invocations.
- Simply providing revisions to a usage pattern when the library evolves is insufficient: A *mapping* is often required between usage pattern versions in order to transform usage pattern instances.
 - While annotating the library is ok—annotating every code point /call site using the library functions along with the arguments needed to call is a drag on developers.
 - The assumption that developers and 3rd party library vendors use the same ontology may not be realistic in the general context.

Machine Learning: We also explored using machine learning to adapt software. We developed a code migration toolchain based on sequence-to-sequence machine learning. This toolchain can train artificial neural networks to detect and repair client code automatically.

Under the hood, the tool learns models of the transformation needed to go from source coded against a current version to source coded against a different version from training examples.

Figure C-2 shows the Long Short Term Memory (LSTM) architecture used for the ROS migration problem. This approach is also promising but faced with significant technical challenges:

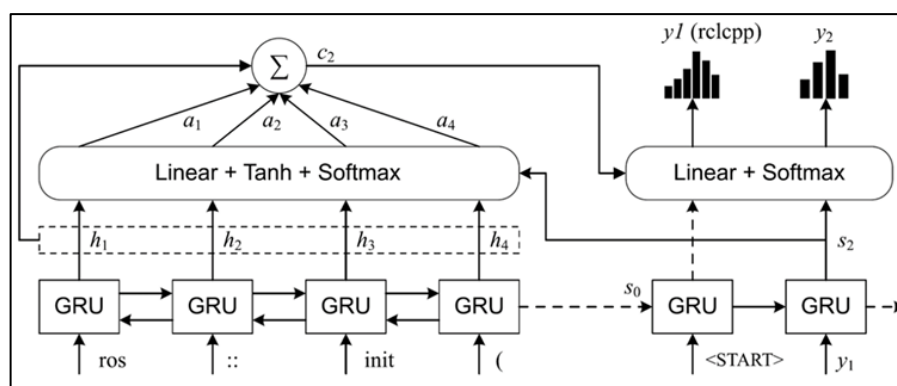


Figure C-2: LSTM model applied to migrate ROS-1 code to ROS-2

- Quality of the translation depends on both quality and quantity of training data. Corpus of curated training data covering the different legitimate possibilities in sufficient numbers is required to train the model. However, finding sufficient number of such before-after samples that are relevant for training seems to pre-suppose a solution to the problem. Training with small samples or incrementally improving the model with reinforced learning or enhancing the model as more translations are available are interesting research direction.
- The neural network model cannot distinguish the important aspects of code migrations from incidental details, like variable names. For example, in the two lines of code shown, the model should focus on learning the structure of the statements, return types, argument types, class/object receiver types etc, and not the form and structure of the variable names, literals etc.

```
JsonParser parser =
    jsonFactory.createJsonParser("{\"name\":\"John\", \" grade\":\"D\"}");

JsonParser jp = jsonFactory.createJsonParser(new File ("user.json"));
```

Pattern-Query based search and transform: In contrast to the prior two approaches, the PQL pattern-query based search and transform approach seeks to develop a robust and easy-to-use means to query and transform code, together with a delivery mechanism integrated with existing

build tools (Maven central repository and plugins). Towards that goal, starting with the initial PQL capability we developed under the base program the technical challenges we addressed and advances we made in the ECP are described in Section 4. The resulting enhanced PQL toolkit has the potential to significantly reduce the burden of API evolution on client code developers.

TECHNICAL ACCOMPLISHMENTS

The pattern-query based approach to mitigate the impact of API evolution has two main technical components: (a) capability to query source code, and (b) capability to transform source code. Within this context, we extended PQL developed in the base effort to support the following enhanced features described below.

Support for code pattern matching involving nesting and hierarchy: A semantic-preserving API evolution may impact a statement inside a construct only if it is enclosed in another construct that satisfies a certain property. Such cases will require PQL to support queries such as

Find uses of `String.concat` occurring in methods that have the `doSomethingWithSomething` signature and only in classes that extend `Human`.

This query should match the use of `concat()` shown in green in the code fragment below, but not the use of `concat()` in red.

```
public class Person extends Human implements Thing {
    public Person () {
        String name = "";
        PersonClient.takesString(doSomethingWithSomething(name.concat("test")));
    }

    public static String doSomethingWithSomething(String
something) {
        return something.concat(Integer.toString(5));
    }
}
```

Supporting this requirement involved verifying additional metadata about the source code in the source code internal representation used by PQL and also extending PQL DSL to support such nesting in the queries.

Ability to specify ordering constraint in the pattern: Depending on the API change, we may need to find a construct that must occur before or after another construct, or it may be the case they need to occur in the matched construct, but their relative order does not matter. It is also possible that the code fragment we are looking for involves a combination of both (i.e., partial ordering). For example, to find a code fragment that declares instances of classes, and uses them in a specific

usage pattern such as “get an instance of a provider, do the work, and then close”, we may not care about the order in which the variables are declared, but do care about the strict sequence of occurrence dictated by the usage pattern. In other words, a query like:

“Find instances where variables of `Person` and `Policy` are declared (in either order) and where `enroll()` is called on a `BenefitsManager` instance after `checkEnrollment()`”

should match both the code fragments shown below:

```
public void
    processDependentBenefits() {
    Person dependent = new Person();
    Policy policy = this.getPolicy();
    BenefitsManager bm=
        BenefitsManager.getInstance();
    if (bm.checkEligibility (this)) {
        bm.enroll(policy, dependent);
    }
    bm.close();
}
```

```
public void
    processDependentBenefits() {
    Policy policy = this.getPolicy();
    Person dependent = new Person();

    BenefitsManager bm=
        BenefitsManager.getInstance();
    if (bm.checkEligibility (this)) {
        bm.enroll(policy, dependent);
    }
    bm.close();
}
```

Supporting this requirement involved enhancing the way PQL searches the internal representation of Java code, also providing a way to indicate in the query where ordering is important or not.

Inferring types of expression in search: An expression can be used as an argument of a function call. This makes it harder to find all matches in a code for queries like:

Find invocations of `String.concat(String value)`

It is common to have calls to `String.concat` with arguments of various non-trivial expressions of type `String` such as:

```
public static String doSomethingWithSomething(String something){
    return something.concat(Integer.toString(5));
}
public static String doAnother() {
    return
doSomethingWithSometing("test").concat(String.valueOf(true));
}
```

```
}
```

Both invocations of `concat()` shown in green above should match. Correctly matching such queries requires resolving symbols and expressions to their language types.

Ability to distinguish context (a statement vs. an expression): The query pattern must also consider the context of the matching code components. For example, we may need to find all invocations of a specific function that occurs in a statement, but not in an expression passed in as an argument. This may be required if invocation of the matched call needs to be preceded by another call due to API evolution, and the calls that appear in a statement and calls that appear in an argument expression need to be handled differently. For example, in the code fragment below, we would like to be able to distinguish between the green and red instances of the `init` calls so that we can transform them differently:

```
public static Something createSomething(Something something){
    if (something == null){
        something = new Something("default");
    }
    return something.init(Integer.toString(5));
}

public static Something createNesting(Something outer) {
    return outer.setInner(createSomething(null).init("inner"));
}
```

Simple user-facing interface: In order to be usable, the query and transformation language must be easy to use and enable lexically preserving code transformations. Using tools and approaches like srcML and Code Property Graph (CPG) *as-is* is problematic in this respect. For instance, srcML represents parse trees as XML, and using that representation for code querying and transformation tasks means relying on XML-based tools and libraries, like XSLT, XPath and XQuery. Expressing complex queries and transformations leads to complex and messy XSLT, XPath and XQuery statements that are not intuitive to Java developers. Similarly, CPG uses a multi-graph to represent a program that includes the Abstract Syntax Tree (AST), the Control Flow Graph (CFG) and the Program Dependency Graph (PDG). This is very powerful, and the graph representation makes querying (i.e., read operation and graph traversal) the internal representation easier, but supporting transformation (i.e., write operation, multi-graph rewriting) becomes very difficult. Consequently, we took the approach to present a Java like DSL for expressing the queries and transformations, and also provided an IDE-like tool to use the PQL toolchain.

Updated PQL Toolchain: PQL addresses each of the challenges described above by providing a query and transform DSL, and an IDE and compilation toolchain that supports hierarchical queries that can check for partial orderings and code context, as well as resolve all symbols and expressions to types.

PQL uses a high-fidelity parse-tree representation of code tailored to Java and capable of enabling format-preserving code transformations. Figure C-3 shows the structure of a simple program in PQL's DSL. The `when` clause of the program finds uses of `String.concat` in methods that return `Strings` defined in classes that extend the `Human` super type. The `*Policy:` declarations express various search directives. The `then` clause in this example directs PQL to simply list the matches found, and does not prescribe any transformation.

```
when {
  fragment(ID:testbinding,FragmentSearchPolicy:RestartAtHome,
    ChildSearchPolicy:RestartAtEndLastMatch) {

    classDeclaration(ID: myClassDeclarationID,
      SuperClassType: org.bbn.pqltest.Human,
      DuplicatePolicy:CycleChildren) {

      methodDeclaration(ID: myMethodDeclarationID,
        ReturnType: java.lang.String) {

        methodInvocation(ID: myMethodInvocationID, ReceiverType: java.lang.String,
          MethodName: concat,
          PatternContext: Statement, ArgumentTypes:java.lang.String,
          DuplicatePolicy: SearchNextUnique);

      }

    }

  }
}
then {
  printMatch(myMethodInvocationID)
}
```

Figure C-3: Illustrative example of a PQL query and transformation in DSL code

The general structure of a program in PQL's DSL is as follows:

```
when {
  fragment (options) {           // Options in this block control matching
                                // policy details such as partial ordering,
                                // duplicates etc

    codePattern (options) {}     // Patterns match code artifacts and bind names
    codePattern (options) {}     // to them

    fragment (options) {}       // Fragments can be nested, allowing us to
                                // write complex queries
  } then {
```

```

        command (options) {}          // Commands act on name bindings resulting from
        command (options) {}          // queries to modify/create/delete code
    }                                  // artifacts

```

```

1  grammar PQL;
2
3  @header {
4
5
6
7  prog:                                WHEN '{' rootFragment '}' THEN '{' commands '}';
8  rootFragment:                       FRAGMENT rootFragmentOptions '{' codeOrFragmentPatterns '}';
9  codeOrFragmentPatterns:             (fragmentPattern | codePattern)+;
10 fragmentPattern:                    FRAGMENT fragmentOptions '{' codeOrFragmentPatterns '}';
11 codePattern:                        referenceSearch | methodInvocation | classDeclaration | methodDeclaration;
12
13 @methodInvocation:                  METHOD_INVOCATION '(' ID ':' bindingDecl
14                                     (' RECEIVER_TYPE ':' receiverType)?
15                                     (' METHOD_NAME ':' methodName)?
16                                     (' PATTERN_CONTEXT ':' patternContext)?
17                                     (' ARGUMENT_TYPES ':' (' argumentTypes ')')?
18                                     (' DUPLICATE_POLICY ':' duplicatePolicy)?
19                                     ')'
20                                     ';'
21
22 @methodDeclaration:                 METHOD_DECLARATION '(' ID ':' bindingDecl
23                                     (' ACCESS_MODIFIER ':' accessModifier)?
24                                     (' STATIC_MODIFIER ':' staticModifier)?
25                                     (' ABSTRACT_MODIFIER ':' abstractModifier)?
26                                     (' FINAL_MODIFIER ':' finalModifier)?
27                                     (' NATIVE_MODIFIER ':' nativeModifier)?
28                                     (' SYNCHRONIZED_MODIFIER ':' synchronizedModifier)?
29                                     (' RETURN_TYPE ':' returnType)?
30                                     (' METHOD_NAME ':' methodName)?
31                                     (' PARAMETER_TYPES ':' (' parameterTypes ')')?
32                                     (' THROWS_TYPES ':' (' throwsTypes ')')?
33                                     (' CHILD_SEARCH_POLICY ':' childSearchPolicy)?
34                                     (' DUPLICATE_POLICY ':' duplicatePolicy)?
35                                     ')'
36                                     '{' codeOrFragmentPatterns? '}'
37                                     ;
38
39 @classDeclaration:                  CLASS_DECLARATION '(' ID ':' bindingDecl
40                                     (' ACCESS_MODIFIER ':' accessModifier)?
41                                     (' STATIC_MODIFIER ':' staticModifier)?
42                                     (' ABSTRACT_MODIFIER ':' abstractModifier)?
43                                     (' FINAL_MODIFIER ':' finalModifier)?

```

Figure C-4: A Portion of the Grammar for PQL’s query and transform DSL

We have created an ANTLR-based grammar (see Figure C-4) custom tailored for querying and transforming code for the PQL DSL. The grammar is similar to Java, which makes it easy to identify and transform Java code.

Figure C-5 described the various components of the PQL toolchain. The PQL user provides the PQL source (i.e., the query and transform program in PQL’s DSL). Using the PQL grammar, the toolchain first creates a parse tree of the PQL source. It then generates and compiles Java source code for the query and transform code dynamically. This results in a distributable (can be in executable form) program that queries and transforms target source code for a given API evolution case.

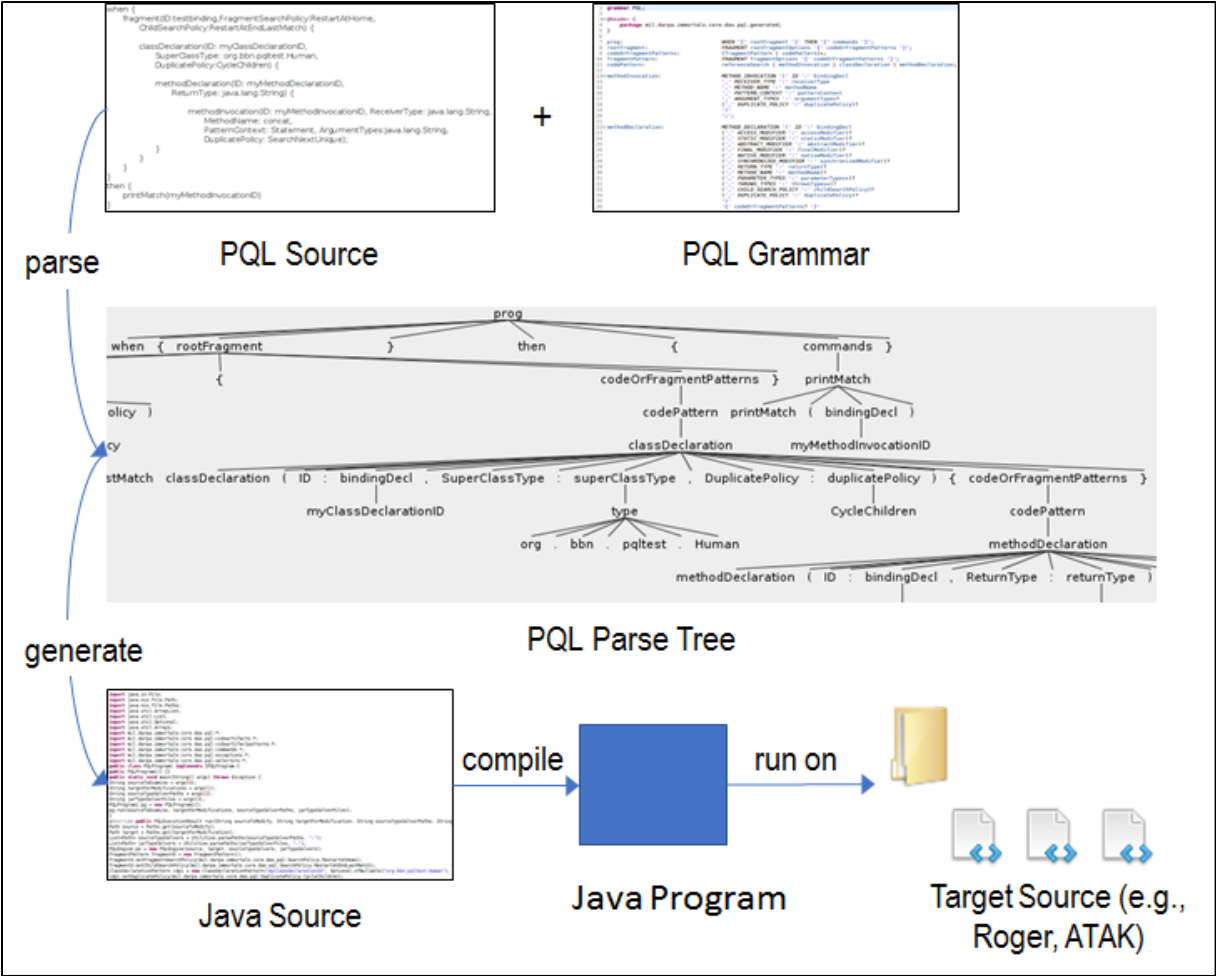


Figure C-5: The PQL toolchain

We envision these programs can be packaged by the library developers with their libraries (for example, using Maven central and Maven plug-ins) when they make new releases with API changes. We did not get to explore the Maven integration in the ECP.

The tool chain is packaged in a PQL IDE (see Figure C-6). The IDE makes it easy to review and approve the changes PQL makes to source code. The user can direct the query and transform program written in PQL's DSL to a single source file or at the entire source tree starting at a root. A before-after code diff is shown in the IDE's center view. Users may approve or reject changes individually or as a whole. A summary of examined and modified files is shown in the lower half.

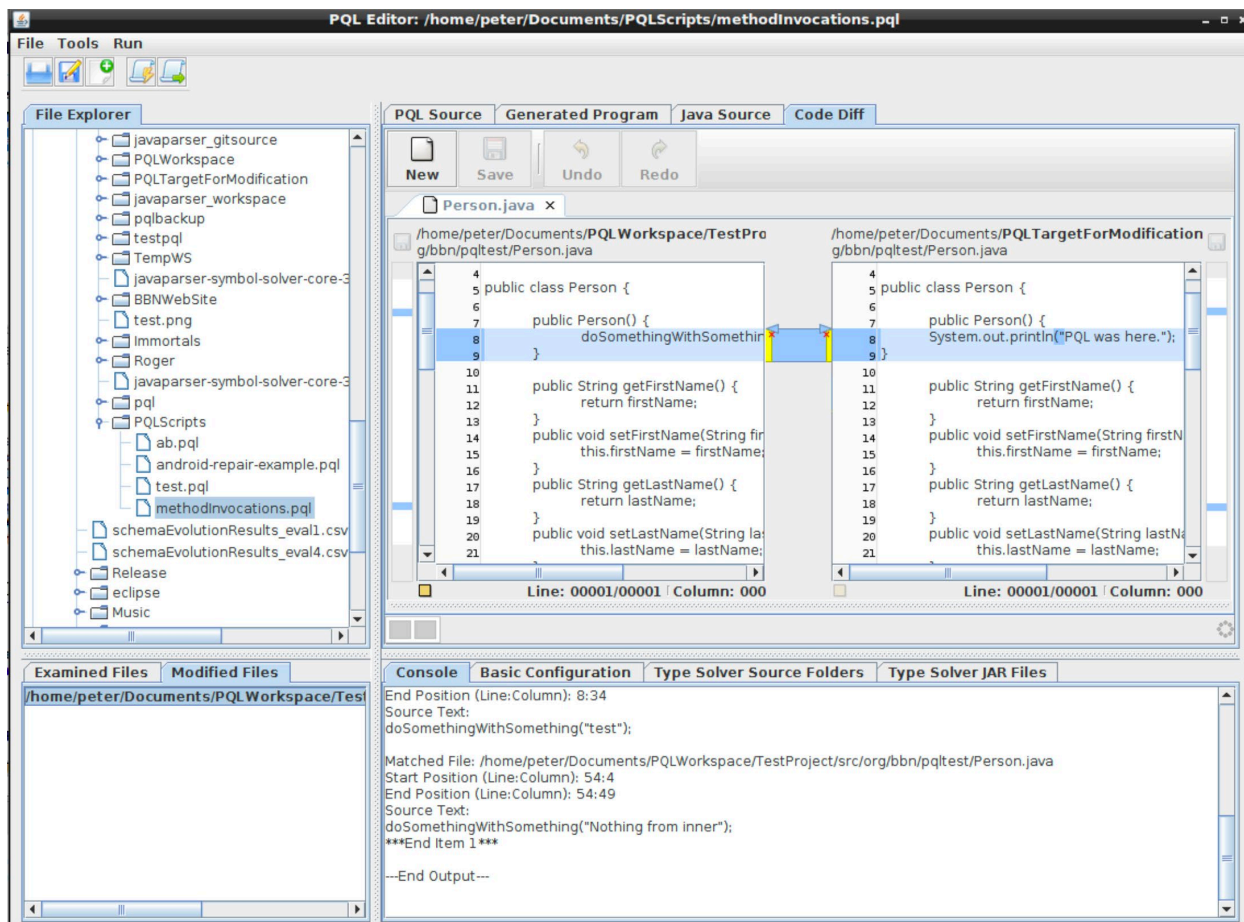


Figure C-6: PQL IDE showing code diff and change review options

RESULTS

Apart from unit tests, we evaluated PQL and the toolchain using isolated API evolution cases on a combination of real and synthetic examples. Tests were successfully executed for each aspect of the PQL grammar described above, including complex cases involving nested class definitions and deep class hierarchies with multiple interface inheritance. In addition, as part of the ECP, we

applied PQL to a real transition use case involving the ATAK-ROGER ecosystem, which we describe in this section in detail.

As mentioned earlier, in this ecosystem there exists a significant number of 3rd party plugins that has dependency on TAK Server and ROGER. For ROGER plugins, each plugin's lifecycle stage, including initialization, is managed by the core ROGER server. Developers noted that providing the plugin with a name during initialization would significantly assist with troubleshooting. Assuming this (i.e., plugins must have a name) in ROGER will break the plugins forcing the plugin developers/maintainers update the plugins. It turned out that the changes required multiple touchpoints that were well defined but time consuming to implement manually.

The changes required to accommodate the name involved the following steps:

- Examine the Java sources files for those that implemented the PlugIn Java interface or one of its descendants.
- Modify the identified files by:
 - Adding a String parameter to the Configure method.
 - Adding a class field to hold the value.
 - Adding a getter and setter method to the file.
 - Adding a method invocation in the Configure method to invoke the setter.

The query part of the PQL source to achieve the required transformation is shown below.

```
when {  
    fragment(ID:testbinding,FragmentSearchPolicy:RestartAtHome,  
            ChildSearchPolicy:RestartAtEndLastMatch) {  
        classDeclaration(ID: classID,  
                        InterfaceTypes: (com.bbn.roger.plugin.Plugin)) {  
            methodDeclaration(ID: methodID,  
                            AccessModifier: Public,  
                            ReturnType: java.lang.Void,  
                            MethodName: configure,  
                            ParameterTypes:  
                                (java.util.Map<java.lang.String,java.lang.Object>,  
                                 com.bbn.roger.plugin.PluginContext)) {}  
        }  
    }  
}
```

The `classDeclaration` codePattern seeks to find classes implementing the Interface type `com.bbn.roger.plugin.Plugin`. The `methodDeclaration` code pattern seeks to identify `configure` methods with the specified signature in the classes identified. It also defines `methodID` and `classID` that binds to each matching context so that the transform part of the PQL source (shown below) can access and refer to them.

```

then {
    printMatch(TargetID: methodID)

    addParameters(ID: addParametersID, TargetID: methodID) {
        (AddAfterPosition: 2, FinalModifier: false, Type: java.lang.String, Name: name)
    }

    addFields(ID: addFieldsID, TargetID: classID, ApplyToAncestorClass: false) {
        (Idempotent: False, AccessModifier: Public, StaticModifier: false, FinalModifier: false,
         Type: java.lang.String, Name: `Context.getStringArray(newParameterNames)[0]`,
         Initializer: null)
    }

    addGetterSetter(ID: addGetterSetterID, TargetID: classID,
        FieldType: java.lang.String,
        fieldName: `Context.getStringArray(newFieldNames)[0]`,
        ApplyToAncestorClass: false, Idempotent: false,
        AddGetter: true, AddSetter: true,
        GetterAccessModifier: Public,
        SetterAccessModifier: Public)

    addMethodInvocation(ID: methodInvocationID, TargetID: methodID,
        Receiver: this, MethodName: `Context.getString(newSetterName)`,
        Arguments: (`Context.getStringArray(newParameterNames)[0]`))
}

```

The directive `addParameter` transforms the method bound to `methodID` to accept a new parameter. The `addFields` directive adds a new class variable to the class bound to `classID`. The `addGetterSetter` directive adds the getter and setter methods for the new field.

PQL identified 176 candidates for the prescribed change. Of these, 80 cases were identified to match the provided pattern. 78 of these cases were modified correctly. The 2 remaining cases require further investigation. In all, 34 kloc source code were analyzed in under 9 seconds (this time included the time required to make repairs to the code). In order for PQL to resolve symbols to their respective types, it must know where the definitions are for these types. These can be provided easily through the PQL UI as source code folders or JAR file locations. If PQL encounters a symbol it cannot resolve, it logs the case for further investigation and continues processing. Our processing encountered 17 such files. Most of these are likely due to external library references that were not present in the evaluation environment (e.g., references to web servers and other dependencies). A future improvement to PQL is to allow configuration of a single root folder for the location of multiple JAR files for type resolution.

CONCLUSION

The current form of PQL with the IDE, and the tool chain with enhancements made to the query and transformation capabilities, resulting from the ECP work represents a usable and extensible prescriptive code transformation tool. The DSL (currently supporting Java) is easy to use, especially for Java developers, so that they (i.e., library developers- who are the originators of the API evolution/subsequent changes in code using the library) can express what needs to be changed in order to update dependent/client code. However, the basic underlying constructs are based on general principles of object-oriented programming, and can be applied to other object-oriented languages as well. Initial transition focused use case showed very good results. We had to spend some time to manually verify the results, but writing the query and transform code to handle the transition case in PQL's DSL, and actually running the tool chain against the target source tree took less than a day's work for a staff engineer. Manually finding the places to change, making the changes and validating by running the tests in the current way typically takes multiple days (depends on the number of files to check and the number of hits).

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AAR	Android ARchive library files
AES	Advanced Encryption System
AFRL	Air Force Research Laboratory
AI	Artificial Intelligence
AM	Adaptation Module
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
ASM	A Java bytecode manipulation and analysis framework
ATAK	Android Team Awareness Kit
ATAK Lite	A lite (simpler) version of Android Team Awareness Kit
BBN	Bolt, Beranek, and Newman; now Raytheon BBN Technologies
BFS	Breadth First Search
BPS	Bits per second
BRASS	Building Resource Adaptive Software Systems
CFG	Context Free Grammer
CHEX	A static analysis method to automatically vet Android apps for component hijacking vulnerabilities
CM	Change Manager
CP	Challenge Problem
CPU	Central Processing Unit
CSV	Comma-Separated Values
DARPA	Defense Advanced Research Projects Agency
DAS	Discovery and Analysis System
DAU	Data Acquisition Unit
DES	Data Encryption Standard
DES3	Triple DES (Data Encryption Standard)
DoD	Department of Defense
DOM	Document Object Model
DSL	Domain Specific Language
EAR	Enterprise Application aRchive
ECP	Engineering Change Proposal
FIPS	Federal Information Processing Standards
FTC	Flight Test Configuration
FTE	Full Time Equivalent
GB	Gigabyte
GME	Generic Modeling Environment
GPS	Global Positioning System
GRU	Gated Recurrent Unit
GVSC	Ground Vehicle Systems Center

HTTP	HyperText Transfer Protocol
ID	XML Identifier
IDAS	Intent-Defined Adaptive Software
IDE	Integrated Development Environment
IDREF	Reference to XML Identifier
IMMoRTALS	Interfaces, Models, and Monitoring for Resource-aware Transformations that Augment Lifecycle of Systems
IO	Input Output
JAR	Java ARchive
JAXB	Jakarta XML Binding (formerly Java Architecture for XML Binding)
JVM	Java Virtual Machine
KB	Knowledge Builder
LINQ	Language INtegrated Query framework (part of Microsoft .NET)
LL	Lincoln Laboratory
LSA	Latent Semantic Analysis
LSTM	Long Short Term Memory
MARTI	Precursor to the Android Team Awareness Kit
MDL	Metadata Description Language
MIT LL	Maschusettes Institute of Technology Licoln Laboratory (LL)
MUSE	Mining and Understanding Software Enclaves
NLP	Non-Linear Programming
OAT	Of Ahead Time- a file format produced by compiling a DEX file with ahead-of-time compilation (AOT)
OOP	Object Oriented Language
OS	Operating System
OSU	Oregon State University
OWL	OWL Web ontology Language
PC	Program Counter
PLA	Product Line Architecture
PLI	Position Location Information
PQL	Program Query Language
QUEMU	Quick EMUlator
RAM	Random Access Memory
R&D	Research and Development
RDF	Resource Description Framework
RNN	Recurrent Neural Network
ROGER	Robust Operational Gateways Enabling Reachback/Reach-forward
ROS	Robot Operating System
ROS1/ROS2	versions 1 and 2 of the Robot Operating System
RTK	RealTime Kinematic
RSA	Rivest Shamir an Adleman (public key cryptosystem)

SA	Situational Awareness
SDK	System Development Kit
SECC	Supersingular Elliptic Curve Cryptography
SEP-95	Spherical Error Probable (SEP-95 = x meters means 95% of points fall within the sphere of radius x centered at the antenna)
SHIF	Class of Description Logic: Attribute language with complex concept negation(S), role hierarchy (H), inverse properties (I), and functional properties- a special case of uniqueness quantification (F)
SHOIN	Class of Description Logic: Attribute language with complex concept negation(S), role hierarchy (H), nominals (O), inverse properties (I), and cardinality restriction (N)
SHOIQ	Class of Description Logic: Attribute language with complex concept negation(S), role hierarchy (H), nominals (O), inverse properties (I), qualified cardinality restriction (Q)
srcML	source M L, a documented oriented XML representation of source code
SMS	Short Message Service
SMT	Satisfiability Modulo Theories
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SROIQ	Class of Description Logic: Attribute language with complex concept negation(S), limited complex role inclusions axioms; reflexivity and irreflexibility; role disjointness (R), nominals (O), inverse properties (I), qualified cardinality restriction (Q)
SW	Software
SwRI	Southwest Research Institute
TA	Technical Area (in the context of the BRASS Broad Agency Announcements)
TARDEC	Tank Automotive Research, Development and Engineering Center (US Army)
UCR	University of California, Riverside
UI	User Interface
URI	Uniform Resource Locator
US	United States
USB	Universal Serial Bus
UTF-8	Unicode Transformation Format – 8-bit
VOA	Voice of the Offense
VM	Validation Manager
WAR	Web Application Resource
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XMLBeans	eXtensible Markup Language Bean
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations