

MOCKINGBIRD: Context-aware Repository-level Code Translation with Isolated Validation

Kaiyao Ke
UC Berkeley
kaiyaoke@berkeley.edu

Ali Reza Ibrahimzada
University of Illinois Urbana-Champaign
alirezai@illinois.edu

Rangeet Pan
IBM
rangeet.pan@ibm.com

Saurabh Sinha
IBM
sinhas@us.ibm.com

Reyhaneh Jabbarvand
University of Illinois Urbana-Champaign
reyhaneh@illinois.edu

Abstract—Despite advances in repository-level code translation, automated type resolution across differing type systems and translation validation (functional-equivalence checking) remain challenging. This paper proposes MOCKINGBIRD, combining context-aware type resolution with mock-based isolated validation. MOCKINGBIRD retrieves API documentation and contextual code information per source-language type, prompting an LLM to resolve precise cross-language type mappings, reducing the manual effort of existing approaches. Using this automatically constructed type map, MOCKINGBIRD employs custom serialization/deserialization to generate equivalent target-language mock objects, validating each translation unit *in isolation* without *test translation*, eliminating test-overfitting risks and translation challenges. On 10 Java-to-Python tasks from the ALPHATRANS benchmark, MOCKINGBIRD improves functional equivalence from the state-of-the-art 25.14% to 43.10%, with no manual type-mapping correction needed. A 700-fragment study shows MOCKINGBIRD achieves 0 false positives and 0.9% false negatives, substantially outperforming GraalVM (2.4% false positives, 7.6% false negatives) and MatchFixAgent (11.0% combined mislabel rate). The pipeline is also LLM-agnostic, maintaining these gains with both DeepSeekCoder-33B and the smaller Devstral-24B.

I. INTRODUCTION

Repository-level code translation is critical for software sustainability and modernization. Existing recent such techniques fall into two categories: neuro-symbolic [1–9] and agentic [10–18]. Neuro-symbolic approaches combine LLMs with program analysis in a fixed pipeline; agentic approaches let LLM agents autonomously explore and translate using tools.

A critical shared component is *translation validation* to ensure functional equivalence with the source. Formal methods do not scale to whole repositories [19]. Test-based validation (translating/executing existing tests [1, 13], differential fuzzing [6, 20]) suffers from: (1) *test coupling effect* [1], which results one bug in dense intra/inter-procedural dependencies invalidates the whole call chain translation; and (2) *test/code overfitting* [21], where LLM-translated tests and code can overfit to each other, passing without true validation.

ALPHATRANS [1] introduced *isolated testing* via GraalVM [22], running source-language tests on individual translated target fragments. While this overcomes *test coupling effect* and *test/code overfitting*, it suffers from (1) *incompleteness* from GraalVM constraints; (2) *lack of*

scalability, requiring semi-automated glue code generation and type mapping; (3) *incompatibility* with automated test generation (GraalVM’s Java version constraints); (4) *false negatives* (e.g., glue-code type mismatches triggering spurious `NullPointerException`); and (5) *false positives* (e.g., undetected global iterator side effects).

MatchFixAgent [11] uses a frontier LLM (Claude Sonnet 3.7) to judge and repair equivalence, avoiding GraalVM pitfalls. This approach is costly. Furthermore, using an LLM in the pipeline for test generation, translation, and validation makes it susceptible to the *test/code overfitting* mentioned earlier, leading to false positives. In fact, of 758 fragments validated by both ALPHATRANS and MatchFixAgent, 233 disagreed, and manual inspection confirmed ALPHATRANS was correct in 62 of them, lower-bounding MatchFixAgent’s error rate at $62/758 = 8.18\%$ (as we will show, on three projects covered by our manual study, MatchFixAgent exhibits an 11.0% mislabeling).

We propose MOCKINGBIRD for *mock-based isolated validation* to overcome the limitations mentioned in prior work. It executes source-language tests, capturing I/O pairs and side effects (field mutations, global/static state changes, control-flow-altering exceptions) per method invocation, then *serializes* this data. A *deserialization* step generates mocks for nested invocations, reconstructing target-language tests for equivalence checking, *without test translation or cross-language execution*. Serialization/deserialization uses *context-aware* type mapping: MOCKINGBIRD retrieves API documentation and code context per type, then uses RAG-based LLM prompting to resolve cross-language type semantics.

MOCKINGBIRD applies to language pairs where (1) the source has good type/API documentation for RAG-based resolution, and (2) the target has a mature mocking framework. We chose Java→Python¹ for evaluation due to their notably divergent, complex type systems, making it a harder setting that better demonstrates MOCKINGBIRD’s power. Furthermore, this translation pair allows us to directly compare with alternative neuro-symbolic and agentic approaches across real-world projects and different techniques for in-isolation

¹Please see §VI for generalizability discussion.

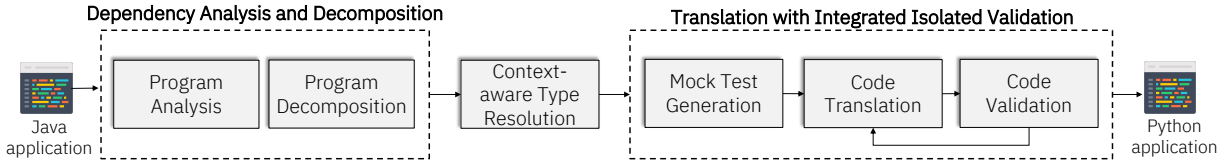


Fig. 1: Overview of MOCKINGBIRD

validation, namely Graal-based validation by ALPHATRANS and unit test generation by MatchFixAgent.

On these ten diverse Java projects (including concurrency-heavy types), MOCKINGBIRD achieves 43.1% functional equivalence versus ALPHATRANS’s 25.14%, with substantially fewer false positives/negatives than GraalVM-based validation. Compared to MatchFixAgent [11], the state-of-the-art LLM-based validation and repair framework, MOCKINGBIRD provides higher coverage, higher reliability, and lower cost. MatchFixAgent’s reported 75.9% equivalence is measured on only 4 of the 10 ALPHATRANS benchmark projects, where ALPHATRANS +GraalVM already achieves 42.6% (vs. 25.1% across all 10 projects). Moreover, its reported 8.18% combined false-positive/false-negative rate is non-negligible. MatchFixAgent also incurs \$1,642 in Claude API costs, on average, only to validate and repair 1,346 translation fragments in four projects, whereas MOCKINGBIRD scales to all 10 projects (4,643 translation fragments) using open-source models (DeepSeekCoder-33B-Instruct and Devstral-24B), achieving only 0.9% combined mislabels without incurring per-token API costs. MOCKINGBIRD’s performance also generalizes across LLMs, including smaller models like Devstral-24B. Our contributions are: (1) a novel automated pipeline for mock-based isolated validation requiring no test translation or cross-language execution, producing equivalent target-language tests; and (2) comprehensive evaluation against state-of-the-art neuro-symbolic and agentic approaches, showcasing the superiority of MOCKINGBIRD.

II. OVERVIEW

Figure 1 presents MOCKINGBIRD’s three main components: *Dependency Analysis and Decomposition* (§III-A), *Context-aware Type Resolution* (§III-B), and *Translation with Integrated Isolated Validation* (§III-C). The first two decompose the source program and resolve types; the third translates and validates decomposed units in isolation, then recomposes the full target-language program.

Dependency Analysis and Decomposition breaks a Java project into field- or method-level translation units (*fragments*), collecting caller/callee dependencies into a JSON *schema* used by later components (§III-A). It also runs existing tests to gather fragment coverage, using EvoSuite to augment unit tests to maximize coverage. *Context-aware Type Resolution* takes the schema and resolves library types from the source to functionally equivalent target types. It crawls source-type API documentation, gathers surrounding code context, and prompts an LLM with this information to predict the equivalent type, validating predictions via a two-step approach (§III-B).

Algorithm 1: Context-aware Type Resolution

Inputs: Subject Projects P_s
Output: Context-aware Type Map CTM

```

1 foreach  $project \in P_s$  do
2    $types \leftarrow \text{getProjectTypes}(project);$ 
3   foreach  $type \in types$  do
4      $doc \leftarrow \text{crawlTypeDocumentation}(type);$ 
5      $code \leftarrow \text{getTypeCodeSnippet}(type);$ 
6      $translation \leftarrow \text{translate\&Validate}(type, doc, code);$ 
7     if  $\text{unsuccessful}(translation)$  then
8        $translation \leftarrow \text{resolveGlobally}(type);$ 
9      $CTM[project][type] \leftarrow translation;$ 
10 return  $CTM$ 

```

Translation with Integrated and Isolated Mock-based Validation takes the type-resolved schema and translates fragments in reverse topological-dependency order using an LLM. MOCKINGBIRD generates mock tests from the source project’s existing tests, isolating each focal fragment by mocking other fragments’ behavior, and builds the harness to run these mock tests in the target PL. After prompting the LLM to translate each fragment and checking syntactic correctness, it executes mock tests to verify functional equivalence; on failure, it iteratively re-prompts the LLM with the failure trace and task context, up to a bounded number of repair attempts (§III-C).

III. MOCKINGBIRD FRAMEWORK

A. Dependency Analysis and Decomposition

Real-world projects exceed LLM context windows; even frontier models that nominally fit a repository degrade before reaching the limit, with cost scaling quadratically with sequence length [23, 24]. Decomposition instead focuses attention on relevant information, reducing noise and hallucinations while improving accuracy [25] and enabling focused debugging. Accordingly, MOCKINGBIRD uses static analysis to decompose projects into *field* and *method* fragments (modifier/s/type/name/value, or signature/body, respectively; application or test). For each, it extracts location, code, dependencies, types, file paths, inheritance, imports, and annotations using Tree-sitter [26] and the Java Call Graph [27], becoming 10× faster than ALPHATRANS’s CodeQL-based approach [28].

B. Context-aware Type Resolution

Automated type resolution is complex [29, 30], addressed previously via symbolic rule-based approaches [31–36] and, more recently, LLMs [37–39]. In code translation specifically, prior work resolves types either purely via LLM (before or during translation) or via neuro-symbolic rules with LLM-guided inference; accuracy here is critical, as incorrect types cause runtime errors that block validation [1, 6].

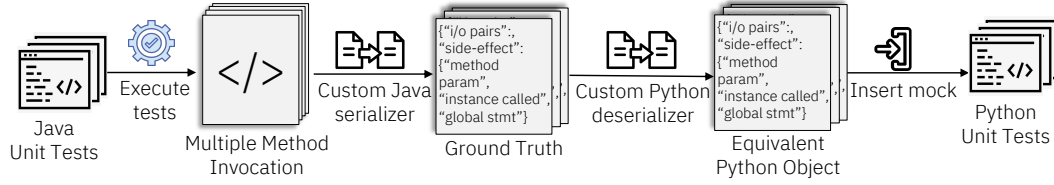


Fig. 2: Mock-based Isolated Validation

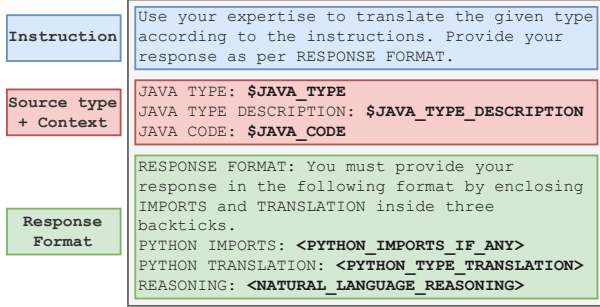


Fig. 3: Context-aware type resolution prompt

Algorithm 1 shows MOCKINGBIRD’s RAG-based, context-aware type resolution workflow. MOCKINGBIRD extracts all source-language types (line 2); application-level types resolve trivially. For library types, it crawls source API documentation, extracts the type’s usage context, and prompts the LLM (Figure 3) to produce target imports, type translation, and natural-language reasoning (lines 3–6). To guard against hallucinated types, MOCKINGBIRD validates the generated imports/translation by generating and executing a Python file that imports and references the type, checking only syntactic/runtime validity. Unresolved translations are inferred from other projects (lines 7–8); if none succeed, the type defaults to object. All translated types and their reasoning are stored in a *Context-aware Type Map* for downstream use (line 9).

Pre-translation resolution with shared reasoning follows from two findings: type-aware pre-conditioning reduces hallucinated/ill-typed API usage versus on-the-fly inference [40, 41], and context-rationalized (not raw) type information improves LLM performance [37, 42]—together improving translation faithfulness and consistency. The same source type can map to different targets by context: Figure 4 shows *commons-cli*’s *Options* class, where both *getMatchingOptions()* and *getRequiredOptions()* return a Java List, but the former builds a modifiable list (warranting Python list), while the latter wraps its result in *Collections.unmodifiableList* (making list typing wrongly mutable). MOCKINGBIRD’s RAG-based resolution correctly infers a tuple for the latter.

Since fully automated resolution (unlike the manual intervention required in ALPHATRANS) cannot guarantee strict semantic equivalence, MOCKINGBIRD’s checks only ensure syntactic/runtime validity. It preserves inferred types in method signatures for structural consistency but does not strictly enforce them in the implementation; instead, predicted types and LLM reasoning serve as contextual guidance, letting the LLM refine type choices based on execution feedback.

```

----- Java List mapped to Python list -----
public List<String> getMatchingOptions(String opt) {
    final List<String> matchingOpts = new ArrayList<>();
    for (final String longOpt : longOpts.keySet()) {
        if (longOpt.startsWith(opt)) {
            matchingOpts.add(longOpt);
        }
    }
    return matchingOpts;
}
----- Java List mapped to Python tuple -----
public List getRequiredOptions() {
    return Collections.unmodifiableList(requiredOpts);
}

```

Fig. 4: MOCKINGBIRD’s different resolved Python types for List depending on the context

C. Translation with Integrated Isolated Validation

MOCKINGBIRD removes back edges from the project’s call graph to make it acyclic, then translates method fragments in reverse topological order. Figure 2 and Algorithm 2 outline the validation workflow: AspectJ-generated execution logs yield I/O behavior and side effects for each invocation in a Java unit test’s call chain (lines 3–4), from which MOCKINGBIRD builds a Python test calling the translated focal method directly (lines 17–18) while mocking all dependent application methods (lines 6–16). Each mock simulates the original method’s return values and side effects (static field updates, exceptions). The test reconstructs initial state, invokes the focal method, and verifies return values, instance/parameter mutations, and global state changes via recursive equality checks (lines 19–26), relying on custom Java object serialization (supporting both standard-library and application types) for reproducible, isolated functional-equivalence validation.

1) *AspectJ-Based Runtime Interception*: An AspectJ instrumentation layer intercepts public method calls, recording IO pairs, side effects, instance-state changes, and static-field modifications, using a broad pointcut over repository packages to exclude synthetic/library calls. On entry, it deep-copies the inputs (via *CustomToStringConverter*), snapshots this for instance methods, and reflects on static fields.

After execution, a second snapshot (parameters, return/exception, receiver state, global state) is compared against the first to detect mutations, serialized into a trace structure for behavior reconstruction and Python test generation. Executions proceed normally post-logging (exceptions re-thrown after recording), and the non-invasive advice layer integrates into Maven test runs without source changes, supporting whole-program or selective instrumentation (Figure 6).

2) *Custom Serialization Workflow*: A *CustomToStringConverter* class serializes Java objects to JSON needed to reconstruct equivalent Python objects, distinguishing appli-

Algorithm 2: Mock-Based Isolated Validation

Input : Java Unit Tests J , Application Method Set R
Output: Mock-Based Python Test Suite T

```

1  $T \leftarrow []$ ;
2 foreach  $test \in J$  do
3    $L \leftarrow \text{runWithAspectJandCustomSerializer}(test)$ ;
4    $focal\_methods \leftarrow \text{extractAllInvocations}(L)$ ;
5   foreach  $focal \in focal\_methods$  do
6      $C \leftarrow \text{getDirectCallees}(focal, R)$ ;
7      $test\_case \leftarrow \text{initTestFile}(focal)$ ;
8     // Mock direct callees of the focal method
9     foreach  $c \in C$  do
10       $M_c \leftarrow \{$ 
11         $output_c,$ 
12         $exception_c,$ 
13         $paramModifications_c,$ 
14         $instanceModifications_c,$ 
15         $staticFieldsModifications_c$ 
16       $\}$ ;
17       $\text{injectMock}(test\_case, c, M_c)$ ;
18      // Set up input of the focal method
19       $S_{init} \leftarrow \{staticFields, focalInstance, focalParams\}$ ;
20       $\text{injectCall}(test\_case, focal, S_{init})$ ;
21      // Verify focal method's output & side effects
22       $Expected \leftarrow \{$ 
23         $output_{focal},$ 
24         $exception_{focal},$ 
25         $paramModifications_{focal},$ 
26         $instanceModifications_{focal},$ 
27         $staticFieldsModifications_{focal}$ 
28       $\}$ ;
29       $\text{injectVerifications}(test\_case, Expected)$ ;
30       $\text{append}(T, test\_case)$ ;
31 return  $T$ 
```

```

byte_array_json = json_obj.get("byte_array", None)
# similar routine to deserialize primitive arrays
byte_array = convert_to_python(byte_array_json)
byte_buffer = bytes([x & 0xFF for x in byte_array])
bytes_io_object = BytesIO(byte_buffer)
bytes_io_object.seek(int(json_obj.get("position", 0)))
```

Fig. 5: Deserialization of Java `ByteArrayInputStream` into Python `io.BytesIO`

cation types (fine-grained introspection) from library classes (grouped handling, e.g., uniform treatment of `Map` implementations) by package prefix. For application classes, the converter introspects fields across the class hierarchy, distinguishing static from instance fields, skipping synthetic/transient members, and recording the declaring class to resolve naming ambiguity. Edge cases get tailored handling: I/O subclasses (e.g., `BufferedReader`, `FilterInputStream`) account for both inherited and custom fields, while inner-class iterators serialize outer references (via synthetic `this$0`) with cached identity tracking, ensuring cycle-free serialization even for deeply nested structures.

This custom serialization enables direct construction of target-language objects, eliminating the need for cross-language interoperability. Unlike `ALPHATRANS`, which relies on manually generated, class-specific glue code with fixed type mappings, our workflow is general-purpose and project-agnostic. `MOCKINGBIRD` supports the Java SE 8 standard library for scalability, leveraging interface-level abstractions to generalize without handling every class individually, e.g., any `java.util.List` implementation (`ArrayList`, `LinkedList`, `Vector`) is handled uniformly via `foreach`

```

@Aspect
class LoggingAspect {
    @Pointcut("project-level main methods") void appPoints() {}
    @Around("appPoints()")
    Object log(ProceedingJoinPoint jp) {
        if (isTestOrSynthetic(jp)) return jp.proceed();
        updateJUnitContext();
        logStart(jp);
        recordInstanceAndArgs(jp);
        recordStaticFieldsBefore();
        Object result = null;
        try { result = jp.proceed(); }
        catch (Exception e) { logException(e); throw e; }
        recordStaticFieldsAfter();
        recordResults(jp, result);
        logEnd(jp);
        return result;
    }
    // helpers: detect JUnit methods, capture static
    // fields, find test method in stack, write logs, ...
}
```

Fig. 6: `MOCKINGBIRD` interception/serialization process

```

private static JSONObject handleByteArrayInputStream(
    ByteArrayInputStream bis, ... ) {
    JSONObject bisDetails = new JSONObject();
    bisDetails.put("type", getTypeName(bis, ...));
    Class<?> bisClass = bis.getClass();
    /* Retrieve underlying byte array */
    Field bufField = bisClass.getDeclaredField("buf");
    bufField.setAccessible(true);
    byte[] array = (byte[]) bufField.get(bis);
    // similar method to handle primitive arrays
    bisDetails.put("byte_array",
        handlePrimitiveArray(array, ...));
    /* Retrieve current position */
    Field posField = bisClass.getDeclaredField("pos");
    posField.setAccessible(true);
    int position = posField.getInt(bis);
    bisDetails.put("position", position);
    return bisDetails;
}
```

Fig. 7: Type handling of `ByteArrayInputStream`

iteration, with the Python deserializer reconstructing the appropriate collection type (per RAG-based type mapping). Third-party or user-defined classes undergo field-level recursive serialization down to primitive or standard-library types. For platform-dependent types (e.g., `java.lang.Thread`), `MOCKINGBIRD` avoids overfitting to language-specific runtime states, instead only checking that the translated function returns a corresponding Python thread.

3) Deserialization and Equality Verification Workflow: `MOCKINGBIRD` reconstructs Python objects from `CustomToStringConverter`-serialized JSON logs via a custom recursive pipeline, resolving each structure's Python class (application classes map to preloaded translated classes; Java types map to standard-library equivalents per context-aware type resolution). Primitives convert directly; compounds (tuples, object fields) use type-specific logic. A global dictionary tracks instantiated objects by serialization-time memory address, preserving reference identity and supporting cyclic references. Figure 5 shows this converting a Java `ByteArrayInputStream` (serialized per Figure 6) into a Python `io.BytesIO`.

For custom classes, deserialization introspects static/instance fields, simulating Java visibility via Python name man-

```

class MockTest(unittest.TestCase):
    @patch('focal_method') @patch('helper_1')
    @patch('helper_2')
    def test_mocking(self, helper_2, helper_1, focal_method):
        # Replace non-focal invocation by mocks
        helper_1.side_effect = SideEffect(method_1_invocation_1,
            method_1_invocation_2, ...)
        helper_2.side_effect = SideEffect(method_2_invocation_1,
            method_2_invocation_2, ...)
        # Set up static fields before method calls
        update_static_fields(json.loads(orig_static_fields_json))
        # construct a Python equivalent of all arguments and the
        # original instance of the Java method invocation
        method_args = list(convert_to_python(json.loads(
            args_initial_json)))
        instance = convert_to_python(json.loads(
            instance_initial_json))
        # call the focal method (call chain replaced by mocks)
        returned_obj = instance.focal_method(*method_args)
        # construct a Python equivalent of final Java instance
        # and check if instance is correct after invocation
        instance_reference = convert_to_python(json.loads(
            instance_final_json))
        self.assertTrue(recursive_equal(instance,
            instance_reference))
        # check if side effects are correct
        self.assertTrue(side_effect_is_correct(json.loads(
            focal_method_side_effect)))
        # construct a Python equivalent of the Java argument list
        # after method invocation and check if it matches Python
        # method invocation
        method_args_reference = convert_to_python(json.loads(
            args_final_json))
        self.assertTrue(recursive_equal(method_args,
            method_args_reference))
        # construct a Python equivalent of the Java returned
        # object and check if it matches the object returned by
        # the Python method invocation
        returned_obj_reference = convert_to_python(json.loads(
            returned_obj_final_json))
        self.assertTrue(recursive_equal(returned_obj,
            returned_obj_reference))

```

Fig. 8: A mock test generated by MOCKINGBIRD

gling (e.g., `private int x in Foo → _Foo__x`); types inheriting from Java standard library classes ignore Java-specific fields. Enums reconstruct via `_name_/_value_`; exceptions use `__new__` to bypass argument enforcement while preserving messages. A recursive equality checker then performs deep semantic comparison, with type-specific logic for primitives, collections, enums, and I/O streams (e.g., `BytesIO/StringIO` via buffer content) and recursive `__dict__` comparison for custom objects. It normalizes numerical tolerances (e.g., `timedelta` precision loss) and string-format differences (e.g., `HashMap` vs. `dict`), ensuring accurate behavioral equivalence while preserving identity, mutability, and visibility semantics.

4) Automated Test Method Generation and Mocking:

MOCKINGBIRD generates mock tests validating each invoked method’s translated Python counterpart against recorded Java behavior, in four steps. At **Reading the AspectJ Log** step, MOCKINGBIRD retrieves the focal method’s I/O/side-effect data (for comparison) and callees’ data (for mock construction), isolating validation. At **Mock Decorator Insertion** step, it inserts decorators distinguishing constructors (`<init>`) from methods, patching each once; the focal method runs directly while others are mocked (simulating output, static-field updates, argument changes, exceptions). **Both public and private methods** are mockable: private methods (e.g., `foo` in `Bar`) become dunder-named (`_Bar__foo`), patchable via

TABLE I: Effectiveness of MOCKINGBIRD (MB) in repository-level code translation compared to ALPHATRANS (AT). **AMF**: #Application Method Fragments, **Non-Validatble**: Fragments where isolated validation cannot validate (no test coverage for both approaches and GraalVM error for ALPHATRANS), **Success/Failure**: Fragments where isolated validation pass/fail (GraalVM for ALPHATRANS and mocking for MOCKINGBIRD).

Subject	AMF	Non-Validatble (%)		Success (%)		Failure (%)	
		AT	MB	AT	MB	AT	MB
cli	273	17.6	10.3	70.7	68.5	11.7	21.3
codec	680	29.1	22.8	38.4	43.7	32.5	33.5
csv	235	34.5	20.9	38.7	49.8	26.8	29.4
exec	248	64.5	48.4	33.5	37.9	2.0	13.7
fast-pfor	748	63.8	36.4	12.0	40.4	24.2	23.3
fileupload	192	90.1	20.8	8.9	66.2	1.0	13.0
graph	530	52.3	24.7	24.8	29.1	22.9	46.2
jansi	409	80.4	47.7	8.1	31.1	11.5	21.3
pool	682	92.7	36.1	6.0	41.9	1.3	22.0
validator	646	58.4	24.5	30.5	48.0	11.2	27.6
Total	4643	59.3	30.0	24.5	43.1	16.2	26.9

`unittest.mock`, avoiding the usual cross-language visibility limitation. At the **Method Body Construction** step, MOCKINGBIRD initializes static fields/instances/parameters to mirror pre-invocation state, invokes the focal method (dependencies mocked), then verifies instance/parameter mutations, static-field changes, return values (recursive comparison), and exceptions against original behavior. Finally, at **File Writing** step, it assembles decorators, parameters, and bodies into a new test file per focal method, enabling isolated verification; runtime object setup/equivalence uses the deserialization workflow above. Figure 8 shows an example mock test. All generated mock tests are publicly available [43].

IV. EVALUATION

To evaluate different aspects of MOCKINGBIRD, we study the following research questions:

RQ1: Effectiveness. To what extent can MOCKINGBIRD effectively translate real-world projects? Does it improve functional equivalence compared to ALPHATRANS?

RQ2: Reliability of Mock-based Isolated Validation. How reliable is mock-based translation validation compared to GraalVM-based validation of ALPHATRANS and unit test generation and translation of MatchfixAgent?

RQ3: Generalizability of MOCKINGBIRD to Other Models. How does MOCKINGBIRD perform when integrated with different LLMs?

RQ4: Contribution of Core Components. Do MOCKINGBIRD’s two key innovations, RAG-based type resolution and mock-based isolated validation, each play a significant role in improving its overall performance?

We use the ALPHATRANS benchmark of ten real-world Java projects, chosen for their diversity and relevance to repository-level translation challenges, particularly mock-based isolated validation. MOCKINGBIRD integrates Tree-sitter [26] and Java Call Graph [27] for static analysis, and Python Mock

TABLE II: Line and branch coverage with existing developer-written tests vs. MOCKINGBIRD’s EvoSuite-augmented test suite. Δ is the absolute percentage-point gain.

Subject	Existing Tests (%)		w/ EvoSuite (%)		Δ (pp)	
	Line	Branch	Line	Branch	Line	Branch
cli	96.25	93.99	99.04	97.13	+2.79	+3.14
codec	91.09	92.19	98.43	96.80	+7.34	+4.61
csv	89.33	82.18	91.64	85.06	+2.31	+2.88
exec	60.83	50.45	81.72	75.11	+20.89	+24.66
fast-pfor	66.09	65.49	69.14	71.66	+3.05	+6.17
fileupload	36.26	44.85	84.52	80.91	+48.26	+36.06
jansi	26.29	20.59	58.01	47.18	+31.72	+26.59
graph	84.93	78.54	98.18	95.38	+13.25	+16.84
pool	33.99	44.44	65.90	64.02	+31.91	+19.58
validator	62.51	65.67	79.56	79.89	+17.05	+14.22
Total	67.44	70.02	79.43	81.47	+11.99	+11.45

5.2.0 [44], JUnit 4/5 [45], Pytest 8.2.1 [46], JaCoCo [47], and Python’s coverage tool [48] for dynamic analysis. We use DeepSeekCoder-33B-Instruct [49] as the primary LLM (matching AlphaTrans, for reproducibility and comparison), and Devstral-24B [50] for ablations showing generalizability. All models use temperature 0 (deterministic output) with default generation parameters otherwise. Following ALPHATRANS’s configuration, MOCKINGBIRD also performs up to four re-prompting attempts, shared between syntax-based re-prompting (triggered by syntax errors) and feedback-based re-prompting (triggered by mock test failures).

A. RQ1: Effectiveness of MOCKINGBIRD

Table I compares MOCKINGBIRD’s effectiveness to ALPHATRANS. MOCKINGBIRD evaluates functional equivalence for 69.98% of AMFs across all 10 projects versus 40.73% for ALPHATRANS, mainly due to MOCKINGBIRD’s EvoSuite generation. Of 1,394 uncovered AMFs, 1,084 are trivial stub/interface methods; excluding these, only 8.71% of non-trivial AMFs (310/3,559) remain uncovered. We exclude LLM-generated tests to preserve determinism. EvoSuite lifts average line coverage from 67.44% to 79.43% and branch coverage from 70.02% to 81.47% (Table II), with *fileupload*, *jansi*, and *pool* gaining over 25 points. Among validatable AMFs, **61.59%** pass all mock tests versus **43.30%** for ALPHATRANS/GraalVM.

Overall, MOCKINGBIRD achieves **43.10%** functional equivalence versus ALPHATRANS’s **24.5%** [49]², driven by evaluating more AMFs, better type resolution, and more rigorous validation (ablation in §IV-D). Normalizing by validatable AMFs, ALPHATRANS has 1,891 validatable AMFs with a **39.87%** failure rate, versus MOCKINGBIRD’s 3,249 validatable AMFs at **38.41%**, lower despite validating 1.72 $\times$ more fragments. This gap widens once GraalVM’s false positives (§IV-B) and stronger-LLM effects (§IV-D) are accounted for.

All translated AMFs scored 10/10 on Black [51], indicating idiomatic output. Table III shows MOCKINGBIRD also produces more passing translated tests despite test translation being less isolated and prone to false positives.

²ALPHATRANS reports 25.14%, the union of GraalVM success (24.50%) and test-translation success (2.88%).

TABLE III: Effectiveness of MOCKINGBIRD (MB) in validating AMFs via direct test translation, compared to ALPHATRANS (AT). ATP: Fragments All Test Pass, OTF: Fragments One Test Fail, MTF: Fragments Many Test Fail, ATF: Fragments All Test Fail, TPR: Test Pass Rate (all translated tests).

Subject	AMF	ATP		OTF		MTF		ATF		TPR	
		AT	MB	AT	MB	AT	MB	AT	MB	AT	MB
cli	273	8.4	11.4	10.6	12.5	32.2	18.7	7.0	13.2	10.1	15.7
codec	680	4.1	2.9	4.4	10.2	12.8	26.8	4.1	17.2	9.4	14.5
csv	235	0.0	6.0	13.6	11.1	0.0	5.5	2.6	10.2	0.0	2.8
exec	248	4.4	5.2	2.0	10.5	7.3	2.8	6.9	11.7	19.3	8.8
fast-pfor	748	4.3	20.5	1.7	8.3	3.7	17.7	3.1	7.9	20.1	28.1
fileupload	192	3.7	11.5	6.8	1.6	1.0	4.7	0.0	3.1	63.4	10.9
graph	530	0.0	2.6	0.9	7.2	0.2	1.1	0.6	2.8	11.0	6.2
jansi	409	0.2	0.0	1.0	0.7	0.0	0.0	0.0	1.7	1.1	0.0
pool	682	1.6	5.3	1.0	9.1	0.3	1.8	0.0	5.0	6.6	12.8
validator	646	3.3	3.7	5.4	4.0	4.2	1.1	4.0	3.6	11.7	4.2
Total	4643	2.9	7.0	3.7	7.5	5.4	9.0	2.6	7.5	9.8	13.1

```
private final Map<String, Option> optionMap
    = new LinkedHashMap<>();
public Collection<Option> getOptions() {
    return optionMap.values();
}
```

Fig. 9: An example of GraalVM Error

Summary. MOCKINGBIRD evaluates and validates functional equivalence of significantly higher AMFs than ALPHATRANS, 43.10% compared to 24.5%.

1) *More Validatable Fragments:* GraalVM-based validation is limited in two ways: across the ten subject projects, 43.43% of AMFs lack unit test coverage, and another 24.50% cannot be validated by GraalVM despite substantial glue-code effort, due to complex types. Uncovered AMFs need automated test generation, but new tests often introduce unseen types. GraalVM’s type-specific glue code requires regenerated, manually adapted bridge code for any type variation (e.g., generics, container element types). Even covered AMFs may trigger unresolvable “GraalVM Errors.” Figure 9 illustrates a host–guest mismatch: `optionMap.values()` returns a `Collection<Option>` view backed by a host-side `LinkedHashMap`, but GraalPython lacks interop support for `java.util.Collection` containing unregistered objects like `Option`. MOCKINGBIRD’s mocking workflow instead intercepts the call at runtime, identifies the `LinkedHashMap` type, and constructs a faithful Python equivalent.

Summary. Mock-based isolated validation scales the number of validatable AMFs by overcoming limitations of GraalVM.

2) *Automated and More Reliable, Context-Aware Type Mapping:* ALPHATRANS reports 91.99% type translation accuracy versus MOCKINGBIRD’s 89.00%—a slight, expected drop from MOCKINGBIRD’s richer context prompting occasionally increasing hallucinations (especially imports). The key difference is scalability: ALPHATRANS needs manual intervention for 8.01% of failures plus correction of 12.24% of LLM-based translations (55 developer-

```

public synchronized void println(final Appendable appendable
)
{
    throws IOException {
        if (getTrailingDelimiter()) {
            appendl(getDelimiterString(), appendable);
        }
        if (recordSeparator != null) {
            appendl(recordSeparator, appendable);
        }
    }
}

```

Fig. 10: An example showcasing effectiveness of context-aware type resolution

```

----- Java Code -----
if (it.hasNext()) {
    buf.append(", ");
}

----- LLM Translation -----
# superfluously advances the iterator
if next(it, None) is not None:
    buf.write(", ")

```

Fig. 11: GraalVM false positive due to API mismatch

hours), while MOCKINGBIRD assigns object to unresolved types with no manual effort, matching accuracy. ALPHATRANS’s fixed mapping can also mislead despite passing runtime checks: in `commons-csv`, `java.nio.CharBuffer` maps to `list[str]`, but Python lists lack buffer semantics and can’t be passed to `TextIOWrapper.readinto()`; MOCKINGBIRD instead infers `bytearray`. Similarly (Figure 10), Java’s `Appendable`—mapped by ALPHATRANS to a context-free `List`—is more plausibly a `StringBuilder` or `PrintWriter` given surrounding delimiter-handling code. MOCKINGBIRD correctly suggests `io.StringIO` in both cases, while ALPHATRANS’s fixed mapping yields erroneous calls like `StringIO("".join(appendable))`.

Summary. Context-aware type resolution eliminates the need for manual effort by ALPHATRANS, enabling more faithful handling of complex types and preserving semantics during code translation.

B. RQ2: Reliability of Mock-based Isolated Validation

RQ1 shows mock-based isolation validates more AMFs, raising the question of reliability: a false negative reports failure for an equivalent translation; a false positive (more problematic) reports success for a non-equivalent one. We manually investigated mock tests and ALPHATRANS’s GraalVM results (plus its human-repair study) across three projects, analyzing translations and validation artifacts for 700 AMFs. These 700 AMFs are **all** AMFs from `cli`, `csv`, and `fileupload`, one project per API-diversity group among the ten benchmark projects: (1) parsing/structured logic (`cli`, `codec`, `validator`, `jansi`); (2) data structures/stream processing (`csv`, `graph`, `pool`); (3) I/O- or OS-intensive workflows (`fileupload`, `fast-pfor`, `exec`). Rather than randomly sampling 700 from 4,643 AMFs, risking project imbalance and reduced reliability, we analyze **all** fragments in

TABLE IV: Manual analysis of translations and mock-based/GraalVM-based/human validation.

Category	GraalVM	Mocking	Human
Success	301 (43.0%)	464 (66.3%)	535 (76.4%)
Failure	97 (13.9%)	146 (20.9%)	165 (23.6%)
Error	102 (14.6%)	0 (0.0%)	-
Not Covered	200 (28.6%)	90 (12.9%)	-
Total	700	700	700
Mislabels	GraalVM	Mocking	-
False Positive	17 (2.4%)	0 (0.0%)	-
False Negative	53 (7.6%)	6 (0.9%)	-

call-graph order from the three chosen projects. Detailed logs are in the MOCKINGBIRD artifact [43].

Each AMF was independently labeled by *three inspectors* (two authors, one external non-author), blinded to validation outcomes, with success requiring majority agreement (2/3). Inter-rater agreement was $\kappa = 0.79$ (Fleiss’ kappa), substantial, with an “almost perfect” threshold of 0.81.

Table IV shows mock-based validation achieving a substantially higher success rate than GraalVM across all three projects, mainly due to producing no errors on complex generic types and validating more AMFs via EvoSuite augmentation. Mocking shows **no false positives** and only **0.9% false negatives**, versus GraalVM’s **2.4% false positives** and **7.6% false negatives**, despite GraalVM validating fewer AMFs overall. Figure 12 shows substantial agreement among all three methods (278 AMFs labeled “translation success” by GraalVM, mocking, and human validation alike). Against human ground truth, mocking validated 186 AMFs GraalVM missed, while GraalVM validated only six cases mocking didn’t cover (17 GraalVM-only successes are later shown to be false positives). Both have unvalidatable cases, but GraalVM has notably more, highlighting its key limitation: dependence on original test-suite coverage.

Summary. Manual analysis of 700 AMFs from three representative projects shows mocking achieves no false positives and only 0.9% false negatives, substantially outperforming GraalVM (2.4% false positives, 7.6% false negatives). On the same projects, MatchFixAgent [11] reports an 11.0% combined mislabel rate despite using a stronger LLM (Claude Sonnet 3.7) than MOCKINGBIRD’s. *Mock-based validation is thus both more accurate and able to validate more fragments via automated test augmentation.*

1) *Comparison with MatchFixAgent:* MatchFixAgent [11], the SOTA LLM-based in-isolation validator, drops language interoperability entirely, using a frontier LLM (Claude Sonnet 3.7 [52]) to judge equivalence directly from the source, candidate translation, and project context. This avoids GraalVM pitfalls but is more expensive and less reliable than mocking. MatchFixAgent reports a cost of \$1.22, on average, to validate an AMF (it assumes the translation exists, and only

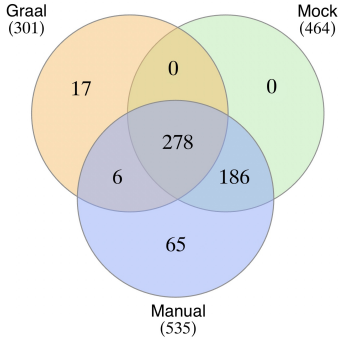


Fig. 12: Success overlap across GraalVM-based, mock-based, and human validation

```

----- Java Code -----
return option == null ? longOption : option;

----- LLM Translation -----
# goes to "else" if option is empty string
return self.__option if self.__option else self.
__longOption

```

Fig. 13: GraalVM false positive due to test incompleteness

repairs if it deems it needed). MatchFixAgent translates only four out of ten ALPHATRANS subjects, i.e., 1,346 AMFs as reported. With the mentioned average cost, this estimates their cost to be around \$1,642, making it costly to scale compared to MOCKINGBIRD’s open-source LLMs (e.g., DeepSeekCoder-33B-Instruct [49], §IV-D), which keeps costs compute-bound rather than per-token.

Reliability-wise, MatchFixAgent suffers from compounding errors due to LLM reasoning uncertainty and shallow test coverage. Of 758 jointly-validatable fragments across those four projects, 233 disagreed, with 62 confirmed in ALPHATRANS’s favor, lower-bounding error at $62/758 = 8.18\%$. *commons-validator* is structurally atypical, ~ 13 near-identical, one-line validator classes wrapping logic in shared *Abstract*Validator* classes, contributing 458/758 (60%) trivially-equivalent fragments. Restricting to *cli*, *csv*, and *fileupload* (our manual study’s projects, with substantive logic) reveals MatchFixAgent’s 11.0% combined mislabel rate (300 jointly validatable, 108 disagreements, 33 confirmed in ALPHATRANS’s favor). MOCKINGBIRD yields no false positives and 0.9% false negatives on the same projects (Table IV), substantially more reliable and cheaper despite MatchFixAgent’s stronger model.

2) *GraalVM False Positives*: Figure 11 shows Java’s `hasNext()` mistranslated as Python’s `next()`. The latter erroneously advances the iterator. GraalVM passes since no test checks iterator state; mocking catches the unnecessary advancement. GraalVM also cannot use EvoSuite tests, relying on developer tests that often miss edge cases. In Figure 13, an empty-string `option` should return `longOption` (non-null), but the Python translation wrongly takes the `else` branch, missed by developer tests, caught by EvoSuite mocks.

3) *GraalVM False Negatives*: GraalVM shows substantial false negatives: glue-code bugs often surface

```

----- Java Code -----
public static OptionBuilder withArgName(final String
    name) {
    OptionBuilder.argName = name;
    return INSTANCE;
}

----- LLM Translation -----
@staticmethod
def withArgName(name: str) -> "OptionBuilder":
    # Throws PolyglotException (bug in glue code)
    OptionBuilder.__argName = name
    return OptionBuilder.__INSTANCE

```

Fig. 14: A translation that results in GraalVM PolyglotException

```

----- Java Code -----
private static void reset() {
    argCount = Option.UNINITIALIZED;
}

----- LLM Translation -----
@staticmethod
def __reset() -> None:
    import Option
    # throws NoClassDefFoundError (bug in glue code)
    OptionBuilder.__argCount = Option.UNINITIALIZED

```

Fig. 15: A translation that results in GraalVM Error (NoClassDefFoundError)

as runtime errors rather than compilation failures, misreporting correct translations as test failures. Figure 14 shows a resulting PolyglotException: `OptionBuilder.withArgName("ICSE")` triggers a Python-to-Java field assignment (`OptionBuilder.__argName = name`) on a private static String, but a latent glue bug fails to expose that field, so GraalVM can’t assign the Python string, raising the exception. Exporting the field (`@Export` or `allowHostAccess=ALL`) would fix this. The error stems from glue-code misconfiguration, not a translation defect.

Filtering GraalVM errors like PolyglotException as false alarms is tempting, but distinguishing translation bugs from glue-code bugs is often hard, especially for exceptions like `NullPointerException` or `NoClassDefFoundError`. Figure 15 shows glue code failing to expose class `Option` (including `UNINITIALIZED`) to the polyglot engine, causing `NoClassDefFoundError` despite the Python file correctly importing `Option`, since GraalVM’s host-guest boundary means Python’s `import` only affects the guest runtime, not the host Java classloader. Even with all glue-code bugs fixed, GraalVM can still fail from inherent interoperability limits: Figure 16 (*commons-cli*) shows correct code and glue logic still raising a runtime exception, since Java’s `contains()` and `remove(index)` operate on indices rather than iterators.

Under GraalVM, Python’s `in` operator uses a `java.util.Iterator`, but `remove(value)` calls the collection’s `remove()` rather than the iterator’s, triggering `java.util.ConcurrentModificationException` due to Java’s fail-fast iterator semantics. The translation is correct, as Python iterators lack modification counts, reflecting

```

----- Java Code -----
if (requiredOpts.contains(key)) {
    requiredOpts.remove(requiredOpts.indexOf(key));
}

----- LLM Translation -----
if key in self.__requiredOpts:
    self.__requiredOpts.remove(key)

```

Fig. 16: A correct translation with Graal Error

```

----- Java Code -----
@Override
public int hashCode() {
    return Objects.hash(longOption, option);
}

----- LLM Translation -----
def __hash__(self) -> int:
    return hash((self.longOption, self.option))

```

Fig. 17: MOCKINGBIRD false negative example

a semantic mismatch in GraalVM’s interop layer, not a translation or glue-code flaw. Other false-negative sources include flaky tests from ALPHATRANS’s assertion-level test decomposition, which introduces order dependencies [53, 54] and spurious failures.

4) *Mocking False Positives*: We found no false positives from mock-based validation, though they can’t be ruled out, particularly where EvoSuite tests miss branch coverage or where side effects that mocking can’t capture directly occur, e.g., remote-server state changes.

5) *Mocking False Negatives*: Mock-based validation can yield rare false negatives when a function lacks deterministic I/O, yet mocking assumes identical outputs across invocations, common with system time, randomness, or hash-based identity. Figure 17 shows the mocking framework wrongly expecting Python’s `hashCode()` equivalent to match a Java-observed integer, despite Python’s differing, run-variable hash algorithm. The translation is correct; only validation fails to account for nondeterminism. EvoSuite tests show similar flakiness when hard-coding environmental assumptions (locale, OS, system time), a broader issue with random test generation, not unique to mocking. Still, mock-based false negatives remain rarer than GraalVM’s.

Summary. GraalVM-based validation suffers from false positives (mostly incomplete tests or API mismatches) and false negatives (mostly glue-code bugs, runtime interop errors, flaky tests). Mock-based validation exhibits few false negatives, primarily due to nondeterministic or environment-dependent behavior.

C. RQ3: Generalizability of MOCKINGBIRD to Other Models

We investigate whether MOCKINGBIRD’s effectiveness is tied to a specific LLM (DeepSeekCoder-33B-Instruct [49]) by replicating RQ1 with a smaller, more recent model, Devstral-24B [50]. Table V shows Devstral-24B consistently scoring higher on most metrics, with comparable overall fidelity. MOCKINGBIRD with Devstral-24B achieves **46.61%**

TABLE V: Comparative Analysis Between DeepSeekCoder-33B-Instruct and Devstral-24B concerning All Test Pass (ATP) and Mock Success.

Subject	DeepSeekCoder-33B		Devstral-24B	
	Mock Suc. (%)	ATP (%)	Mock Suc.(%)	ATP (%)
cli	68.50	11.36	78.02	19.05
codec	43.68	2.94	49.26	10.88
csv	49.79	5.96	56.17	10.64
exec	37.90	5.24	56.45	12.50
fast-pfor	40.37	20.45	35.70	10.16
fileupload	66.15	11.46	70.83	16.15
graph	29.06	2.64	31.70	12.26
jansi	31.05	0.00	36.43	23.72
pool	41.94	5.28	43.70	10.85
validator	47.99	3.72	50.46	12.07
Total	43.10	7.04	46.61	12.99

functional equivalence versus DeepSeekCoder-33B-Instruct’s 43.10%, outperforming it on mock success rate in nine of ten projects. More notably, Devstral-24B achieves a 12.99% all-test-pass (ATP) rate for translated tests versus 7.04% for DeepSeekCoder-33B-Instruct (nine of ten projects), which is significant, since test translation, dependent on library semantics and implicit I/O/environment assumptions, is typically harder than source translation.

This performance on a medium-sized open-source model suggests MOCKINGBIRD’s effectiveness does not critically depend on the underlying LLM, consistent with its design: RAG-based in-context type resolution and mock-based isolated validation aren’t optimized for any specific model. Type resolution is relatively lightweight (recognizing type semantics from context rather than deep reasoning), suiting smaller LLMs, while mock-based validation is LLM-independent.

Summary. MOCKINGBIRD’s context-aware type resolution and heuristic-based validation generalize well across different models.

D. RQ4: Contribution of Core Components (Ablations)

1) *Mocking*: We run two ablations. First, we remove all EvoSuite-generated tests to isolate test-augmentation’s contribution to validating otherwise-uncovered AMFs. Second, we disable on-the-fly mock-based validation during translation, relying only on translated developer-written tests and syntactic/runtime checks, running mock-based validation just once after translation completes, testing whether coverage alone suffices. We exclude GraalVM from this ablation since it requires labor-intensive manual glue code for automatically generated tests [1]; RQ1 already shows mocking’s superiority over GraalVM.

Columns 3–4 of Table VI show both ablations substantially degrade performance. Without EvoSuite, equivalence drops from **43.10%** to **29.72%** (still above ALPHATRANS’s **25.14%**), mainly hurting low-coverage projects (*fileupload*, *fast-pfor*) while high-coverage ones (*cli*, *csv*) stay stable, suggesting the gap reflects LLM nondeterminism more than coverage. Without on-the-fly mocking, equivalence drops to **29.27%**, as re-prompting relies on less-reliable, non-isolated

TABLE VI: Effectiveness of MOCKINGBIRD (**MB**) components on validation results. **ES**: EvoSuite, **OTF**: Mocking On-the-fly, **RAG-TR**: RAG Type Resolution).

Subject	MOCKINGBIRD	w/o ES	w/o OTF	w/o RAG-TR
cli	68.50	69.96	12.82	72.89
codec	43.68	35.44	30.74	38.53
csv	49.79	50.21	43.40	40.85
exec	37.90	34.68	22.58	48.39
fast-pfor	40.37	10.29	41.31	37.83
fileupload	66.15	27.60	44.79	59.90
graph	29.06	21.13	32.08	32.26
jansi	31.05	11.49	39.61	28.85
pool	41.94	28.89	25.66	42.82
validator	47.99	39.94	5.42	39.63
Total	43.10	29.72	29.27	41.18

translated tests; simple-test projects (*jansi*, *fast-pfor*) are unaffected, but complex-test projects (*cli*, *validator*) drop sharply, since noisy test translation can reject correct translations or accept faulty ones. This confirms MOCKINGBIRD’s gains come from both coverage and precise per-invocation feedback, enabled by direct object construction and isolated mocking, unlike GraalVM’s coarser validation.

Summary. Mocking contributes to MOCKINGBIRD’s performance both by supporting EvoSuite-based test augmentation and on-the-fly validation with meaningful feedback, demonstrating that increased coverage combined with precise, per-invocation feedback drives reliable and informative translation validation beyond what existing approaches provide.

2) *Context-aware Type Resolution*: Reverting to ALPHATRANS’s fixed, manually curated type mappings drops equivalence from **43.10%** to **41.18%** (Column 5), a modest gap given MOCKINGBIRD’s resolution is fully automated versus ALPHATRANS’s 55-human-hour, 360-type manual curation. MOCKINGBIRD needs no manual intervention, defaulting ambiguous types to `object` while remaining scalable.

3) *Ablation on I/O Only vs. Full Snapshot Validation*: Against Oxidizer’s relaxed criterion (ignoring side-effect/state mutations), I/O-only validation yields 2,151 passes versus full snapshots’ 2,001; the extra 150 are all **false positives** masking incorrect side effects, a 3.23% false-positive rate.

Summary. Ignoring side effects introduces additional false positives (3.23%), demonstrating that MOCKINGBIRD’s full-state checking is necessary to reliably detect incorrect method translations beyond mere input/output equivalence.

V. RELATED WORK

Recent work has explored the use of LLMs for code translation [11, 55–60], showing promising results on synthetic benchmarks but limited effectiveness on real-world software projects. Other studies have similarly employed language models for code migration, with a predominant focus on curated examples [61, 62]. Existing work has also addressed

repository-level translation targeting different language pairs. AlphaTrans [1] translates Java to Python using open-source LLMs and GraalVM [22] for isolated validation. Several techniques translate C to Rust [3, 63, 64]. Oxidizer [6] uses fuzzing and FFI-based interoperability, replacing incorrectly translated functions with delegating wrappers rather than isolating per-function validation. It *ignores* instance-state mutations, parameter side effects, and changes to global/static fields, thereby inviting false positives/negatives. Several methods also integrated transpiler outputs to guide LLM-based translation [19]; agentic pipelines [11, 17, 18] are now capable of translating repositories. Nitin et al. [65] capture natural language specifications from source code to inform translation, while Yang et al. [66] utilize test cases to support the process.

Compared to these techniques, MOCKINGBIRD addresses important limitations in test translation, which many existing techniques rely on to validate the functional equivalence of translations. Our performance is superior to state-of-the-art neurosymbolic and agentic pipelines, notably decreasing false positives and false negatives.

VI. THREATS TO VALIDITY

External Validity. MOCKINGBIRD’s mechanism is *language-pair agnostic*, relying on *execution-driven semantic serialization and reconstruction via type mapping*. Java→Python is a particularly challenging static-to-dynamic case, so less-divergent pairs (e.g., Java↔Kotlin, Go→Rust) should generalize more easily, given the required unit-test and mocking infrastructure is widely available (Python, Rust, Kotlin, C#, JavaScript).

For sparsely documented libraries, LLM-generated synthetic specifications could replace RAG-retrieved documentation without altering the workflow. Runtime introspection transfers readily to other reflective or dynamic languages (C#, Kotlin, JavaScript, Ruby), needing extra instrumentation only for lower-level languages (C/C++, Rust). Deserialization into languages like Rust may need structural adaptations (e.g., `Rc<RefCell<T>`), but this is engineering effort analogous to MOCKINGBIRD’s existing Java→Python normalization, not a fundamental barrier.

Internal Validity. Risks from incorrect mocks or EvoSuite undercoverage are mitigated by manual inspection of 700 AMFs (confirming no false positives and fewer false negatives than GraalVM), an unbiased project pool guarding against project overfitting, and RQ3’s cross-LLM (Devstral) results ruling out model overfitting.

Construct Validity. MOCKINGBIRD relies on well-vetted tools: Tree-sitter [26], JaCoCo [47], Python coverage [48], and Python mock [44].

VII. CONCLUSION

The paper introduced MOCKINGBIRD, a framework that integrates context-aware type resolution with mock-based, isolated validation to achieve high-quality, semantically faithful translations from Java to Python. One promising direction for future work is integrating more fine-grained semantic analysis,

such as control-flow or data-flow reasoning, to further enhance type inference and translation coherence. Another direction for future work is to make the MOCKINGBIRD pipeline PL-agnostic or to enable adaptation to more language pairs with minimal human effort.

VIII. DATA AVAILABILITY

Our artifacts are publicly available at [43].

REFERENCES

- [1] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand, “Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation,” *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: <https://doi.org/10.1145/3729379>
- [2] V. Nitin, R. Krishna, L. Lemos do Valle, and B. Ray, “C2SaferRust: Transforming C projects into safer Rust with neurosymbolic techniques,” arXiv preprint arXiv:2501.14257, 2025.
- [3] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen, “Syzygy: Dual code-test C to (safe) Rust translation using LLMs and dynamic analysis,” in *LLM4Code*, 2025.
- [4] C. Wang, T. Yu, J. Wang, D. Chen, W. Zhang, Y. Shi, X. Gu, and B. Shen, “EVOC2RUST: A skeleton-guided framework for project-level C-to-Rust translation,” in *ICSE-SEIP*, 2026.
- [5] X. Cai, J. Liu, X. Huang, Y. Yu, H. Wu, C. Li, B. Wang, I. N. B. Yusuf, and L. Jiang, “RustMap: Towards project-scale C-to-Rust migration via program analysis and LLM,” in *ICECCS*, 2025, pp. 283–302.
- [6] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening, “Scalable, validated code translation of entire projects using large language models,” in *PLDI*, 2025.
- [7] T. Zhou, Z. Zhang, H. Lin, S. Jha, M. Christodorescu, K. Levchenko, and V. Chandrasekaran, “SACTOR: LLM-driven correct and idiomatic C to Rust translation with static analysis and FFI-based verification,” arXiv preprint arXiv:2503.12511, 2025.
- [8] B. Wang, T. Li, R. Li, U. Mathur, and P. Saxena, “Program skeletons for automated program translation,” in *PLDI*, 2025, pp. 920–944.
- [9] F. Luo, K. Ji, C. Gao, S. Gao, J. Feng, K. Liu, X. Xia, and M. R. Lyu, “Integrating rules and semantics for LLM-based C-to-Rust translation,” in *ICSME Industry Track*, 2025.
- [10] H. Sim, H. Cho, Y. Go, S. A. Kazemi Zarkouei, Z. Fu, A. Shokri, and B. Ravindran, “Search-based multi-trajectory refinement for safe C-to-Rust translation with large language models,” arXiv preprint arXiv:2505.15858, 2025.
- [11] A. R. Ibrahimzada, B. Paulsen, R. Jabbarvand, J. Dodds, and D. Kroening, “MatchFixAgent: Language-agnostic autonomous repository-level code translation validation and repair,” in *Proceedings of the 43rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, T. Zhang, M. Dudik, M. Jaggi, A. Agarwal, S. Li, D. Schuurmans, J. Zhu, F. Berkenkamp, H. Dong, and A. Bietti, Eds., vol. 306. PMLR, 06–11 Jul 2026, pp. 49428–49451. [Online]. Available: <https://proceedings.mlr.press/v306/ibrahimzada26a.html>
- [12] Z. Yuan, W. Mao, Z. Chen, X. Shang, C. Wang, Y. Lou, and X. Peng, “Project-level C-to-Rust translation via pointer knowledge graphs,” in *FSE*, 2026.
- [13] A. Khatry, R. Zhang, J. Pan, Z. Wang, Q. Chen, G. Durrett, and I. Dillig, “CRUST-Bench: A comprehensive benchmark for C-to-safe-Rust transpilation,” in *COLM*, 2025.
- [14] Y. Wang, R. Ou, Y. Wang, M. Liu, J. Chen, E. Shi, X. Liu, Y. Ma, and Z. Zheng, “EffiReasonTrans: RL-optimized reasoning for code translation,” arXiv preprint arXiv:2510.18863, 2025.
- [15] T. Li, R. Li, B. Wang, B. Paulsen, U. Mathur, and P. Saxena, “Adversarial agent collaboration for correctness improvements of C to safe Rust translation,” arXiv preprint arXiv:2510.03879, 2025.
- [16] Z. Guan, X. Yin, Z. Peng, and C. Ni, “RepoTransAgent: Multi-agent LLM framework for repository-aware code translation,” arXiv preprint arXiv:2508.17720, 2025.
- [17] A. R. Ibrahimzada, B. Paulsen, D. Kroening, and R. Jabbarvand, “Recodeagent: A multi-agent workflow for language-agnostic translation and validation of large-scale repositories,” in *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’26. New York, NY, USA: Association for Computing Machinery, 2026, p. 2409–2421. [Online]. Available: <https://doi.org/10.1145/3832783.3837486>
- [18] M. Farrukh, B. Coskun, T. Palit, and M. Polychronakis, “ORBIT: Guided agentic orchestration for autonomous C-to-Rust transpilation,” arXiv preprint arXiv:2604.12048, 2026.
- [19] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, “VERT: Verified equivalent Rust transpilation with large language models as few-shot learners,” in *ASE*, 2025.
- [20] Y. Bai and T. Palit, “RustAssure: Differential symbolic testing for LLM-transpiled C-to-Rust code,” in *ASE*, 2025.
- [21] Y. Chen, A. Ahmad, Y. Zhou, and R. Jabbarvand, “Unlocking model potentials through adaptive multi-agent scaffolding for efficient issue resolution,” arXiv preprint arXiv:2606.25514, 2026.
- [22] Oracle, “GraalVM,” 2025. [Online]. Available: <https://www.graalvm.org>
- [23] S. Chen, S. Wong, L. Chen, and Y. Tian, “Extending context window of large language models via positional interpolation,” arXiv preprint arXiv:2306.15595, 2023.
- [24] IBM Research, “How large can context windows go? Understanding limits of long-context LLMs,” 2024. [Online]. Available: <https://research.ibm.com/blog/>

larger-context-window

- [25] Z. Wang, C. Gao, C. Xiao, Y. Huang, S. Si, K. Luo, Y. Bai, W. Li, T. Duan, C. Lv *et al.*, “Document segmentation matters for retrieval-augmented generation,” in *ACL Findings*, 2025, pp. 8063–8075.
- [26] The Tree-sitter Team, “Tree-sitter,” 2025. [Online]. Available: <https://pypi.org/project/tree-sitter/>
- [27] G. Gousios, “Java call graph,” 2025. [Online]. Available: <https://github.com/gousiosg/java-callgraph>
- [28] GitHub, “CodeQL,” 2025. [Online]. Available: <https://codeql.github.com>
- [29] G. Guizzo, J. M. Zhang, F. Sarro, C. Treude, and M. Harman, “Mutation analysis for evaluating code translation,” *Empirical Software Engineering*, vol. 29, no. 1, p. 19, 2024.
- [30] A. A. Terekhov and C. Verhoef, “The realities of language conversions,” *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2000.
- [31] D. Broggi and Y. Liu, “On the interoperability of programming languages via translation,” in *CSCE*, 2023, pp. 2579–2585.
- [32] D. George, P. Girase, M. Gupta, P. Gupta, and A. Sharma, “Programming language inter-conversion,” *International Journal of Computer Applications*, vol. 1, no. 20, pp. 68–74, 2010.
- [33] K. Lano and H. Siala, “Using model-driven engineering to automate software language translation,” *Automated Software Engineering*, vol. 31, no. 1, p. 20, 2024.
- [34] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, “Unsupervised translation of programming languages,” in *NeurIPS*, 2020, pp. 20 601–20 611.
- [35] E. J. Coco, H. A. Osman, and N. I. Osman, “JPT: A simple Java-Python translator,” *Computer Applications: An International Journal*, vol. 5, 2018.
- [36] L. Qiu, “Programming language translation,” Cornell University, Tech. Rep., 1999.
- [37] H. Dong, J. Yang, X. Deng, Y. Jiang, G. Pekhimenko, F. Long, and X. Si, “TypyBench: Evaluating LLM type inference for untyped Python repositories,” in *ICML*, 2025.
- [38] C. Wang, J. Zhang, R. Wu, and C. Zhang, “Dainfer: Inferring API aliasing specifications from library documentation via neurosymbolic optimization,” in *FSE*, 2024, pp. 2469–2492.
- [39] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, “Generative type inference for Python,” in *ASE*, 2023, pp. 988–999.
- [40] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, “On mitigating code LLM hallucinations with API documentation,” in *ICSE-SEIP*, 2025, pp. 237–248.
- [41] C. Wang, G. Qiu, X. Gu, and B. Shen, “API-RAT: Integrating multi-source API knowledge for enhanced code translation with LLMs,” arXiv preprint arXiv:2504.14852, 2025.
- [42] A. P. S. Venkatesh, S. Sabu, A. M. Mir, S. Reis, and E. Bodden, “The emergence of large language models in static analysis: A first look through micro-benchmarks,” in *FORGE*, 2024, pp. 35–39.
- [43] “MockingBird Artifacts,” 2025, <https://github.com/mockingbird-artifacts/MockingBird>.
- [44] The Mock Python Team, “Mock library Python,” 2025. [Online]. Available: <https://pypi.org/project/mock/>
- [45] The JUnit Team, “JUnit,” 2025. [Online]. Available: <https://junit.org/junit5/>
- [46] pytest-dev, “Pytest,” 2024. [Online]. Available: <https://www.pytest.org>
- [47] The JaCoCo Team, “Java code coverage,” 2025. [Online]. Available: <https://www.eclemma.org/jacoco/>
- [48] N. Batchelder, “Coverage.py,” 2025. [Online]. Available: <https://pypi.org/project/coverage>
- [49] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, “DeepSeek-Coder: When the large language model meets programming — the rise of code intelligence,” arXiv preprint arXiv:2401.14196, 2024.
- [50] Mistral AI and All Hands AI, “Devstral: A new state-of-the-art open model for coding agents,” 2025. [Online]. Available: <https://www.all-hands.dev/blog/devstral-a-new-state-of-the-art-open-model-for-coding-agents>
- [51] Python Software Foundation, “Black: The uncompromising Python code formatter,” 2025. [Online]. Available: <https://github.com/psf/black>
- [52] Anthropic, “Claude 3.7 Sonnet,” 2025. [Online]. Available: <https://www.anthropic.com/news/claude-3-7-sonnet>
- [53] F. Palomba and A. Zaidman, “Notice of retraction: Does refactoring of test smells induce fixing flaky tests?” in *ICSME*, 2017.
- [54] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. Ernst, and D. Notkin, “Empirically revisiting the test independence assumption,” in *ISSTA*, 2014.
- [55] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, “Code-TransOcean: A comprehensive multilingual benchmark for code translation,” in *EMNLP Findings*, 2023, pp. 5067–5089.
- [56] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang, “Rectifier: Code translation with corrector via LLMs,” arXiv preprint arXiv:2407.07472, 2024.
- [57] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, “On the evaluation of neural code translation: Taxonomy and benchmark,” in *ASE*, 2023, pp. 1529–1541.
- [58] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, “Lost in translation: A study of bugs introduced by large language models while translating code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3639226>
- [59] P. Di, J. Li, H. Yu, W. Jiang, W. Cai, Y. Cao, C. Chen, D. Chen, H. Chen, L. Chen *et al.*, “CodeFuse-13B: A pretrained multi-lingual code large language model,” in

ICSE-SEIP, 2024, pp. 418–429.

- [60] S. Tipirneni, M. Zhu, and C. K. Reddy, “StructCoder: Structure-aware transformer for code generation,” *Transactions on Knowledge Discovery from Data*, vol. 18, pp. 1–20, 2024.
- [61] P. Irwin, “Java to CSharp converter,” 2025. [Online]. Available: <https://github.com/paulirwin/JavaToCSharp>
- [62] M. Zhu, K. Suresh, and C. K. Reddy, “Multilingual code snippets training for program translation,” in *AAAI*, vol. 36, 2022, pp. 11 783–11 790.
- [63] M. Shiraishi, Y. Cao, and T. Shinagawa, “Smartc2rust: Iterative, feedback-driven c-to-rust translation via large language models for safety and equivalence,” *arXiv preprint arXiv:2409.10506*, 2024.
- [64] S. Dehghan, T. Sun, T. Wu, Z. Li, and R. Jabbarvand, “Translating large-scale C repositories to idiomatic Rust,” *arXiv preprint arXiv:2511.20617*, 2025.
- [65] V. Nitin, R. Krishna, and B. Ray, “SpecTra: Enhancing the code translation ability of language models by generating multi-modal specifications,” *arXiv preprint arXiv:2405.18574*, 2024.
- [66] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, “Exploring and unleashing the power of large language models in automated code translation,” in *FSE*, 2024, pp. 1585–1608.