Exploratory, Omniscient, and Multiverse Diagnostics in Debuggers for Non-Deterministic Languages

Damian Frölich
dfrolich@acm.org
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

Tommaso Pacciani
t.c.pacciani@uva.nl
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
Informatics Institute, University of Amsterdam
Amsterdam, the Netherlands

Abstract

Debugging non-deterministic programs is inherently difficult as the compound effects of non-deterministic execution steps is hard to predict and gives rise to a (potentially) vast space of reachable program states such that manual exploration of all reachable states is infeasible.

Multiverse debugging addresses these problems by realising a fine-grained, exhaustive and interactive process for state space exploration. At SLE2023, Pasquier et al. presented a generic framework that makes exploration practical through user-defined reductions on program states and by proposing expressive logics for defining and searching for states and traces of interest, generalising the concept of breakpoint. The framework has been validated through the case study language AnimUML designed to make non-deterministic UML specifications executable.

In this paper, we perform additional case studies to evaluate the applicability of the framework. We analyse three non-deterministic, domain-specific languages representing three different domains: grammar engineering, formal operational semantics, and norm engineering. The framework is evaluated against requirements extracted from these domains, resulting in the identification of several limitations of the framework. We then propose a modified and extended framework and apply it to develop multiverse debuggers for the case study languages. The result demonstrates a multiverse debugging framework with more general applicability.

CCS Concepts: • **Software and its engineering** → Parsers; *Domain specific languages*; **Software testing and debugging**.

SLE '25, June 12–13, 2025, Koblenz, Germany

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Keywords: exploratory programming,debuggers,multiverse debuggers,parsing,domain-specific languages

ACM Reference Format:
Damian Frölich, Tommaso Pacciani, and L. Thomas van Binsbergen. 2025. Exploratory, Omniscient, and Multiverse Diagnostics in Debuggers for Non-Deterministic Languages. In *Proceedings of ACM SIGPLAN International Conference on Software Language Engineering (SLE '25).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Conventional **stepwise debuggers** can be used to explore the execution of a program (a run) in a step-by-step manner, giving programmers control to interrupt and proceed execution as they see fit, and enabling them to inspect concrete information about that run at the moment of interruption (e.g., active bindings, variable assignments, object state). The level of granularity of the steps, the control mechanisms, and the observable context information depends per language, but typically involves setting breakpoints on program locations and monitoring mutations to specific variables.

Omniscient debuggers (also referred to as 'back-in-time debuggers') extend stepwise debuggers by recording information during a debug session to allow programmers to revisit some or all steps of the execution [4, 14]. This functionality is particularly useful to understand how a particular (undesirable) program state came to be by retracing steps and the evolution of variables and (other) objects (without having to anticipate meaningful intermediate states beforehand by setting up breakpoints and monitors).

Non-deterministic programs admit multiple runs, each of which can (potentially) exhibit different desirable or undesirable behaviour (bugs). Debugging non-deterministic programs is inherently challenging as the set of possible runs may be vast, hard to predict, and may contain runs that occur only rarely. Conventional debuggers provide only a partial view on the bugs a program admits as they perform a single run per debugging session. Repeated debugging sessions are required and rare bugs may remain unobserved.

To address these challenges, **multiverse debuggers** provide an interactive, user-controlled and simultaneous exploration of multiple runs, avoiding redundant work by detecting syntactically equal states [15]. Pasquier et al. [20]

introduced user-defined reductions over states, giving the user a mechanism to reduce the explored state space. The authors define a language-parametric framework for obtaining omniscient, multiverse debuggers through the definition of a transition relation for the object language. In [21], the authors introduce a generic breakpoint-specification mechanism and demonstrate various logics (such as regular expressions and linear temporal logic) for expressing traces of interest and further inspection.

Exploratory programming [12, 24, 26] is a style of programming in which programmers experiment with code to simultaneously discover the desired result(s) and the (preferred) way of reaching the result(s). In [29], exploratory programming is formulated as an extension of incremental (REPL-based) programming in which a program is developed incrementally by submitting program fragments and observing their effects. Exploratory programming generalises incremental programming in a way similar to how multiverse debugging generalises stepwise debugging: multiple runs are explored interactively and in parallel in order to investigate (un)desirable outcomes. In exploratory programming, a programmer may additionally be interested in directly comparing the effects of multiple runs.

In this work, we introduce the execution graph to an extended version of the framework of Pasquier et al. and generalise the concepts of breakpoint and reduction to admit additional user scenarios inspired by exploratory programming. We evaluate the applicability of multiverse debugging and the generality of the original and extended framework by using three case studies: grammar exploration, formal operational semantics, and reasoning with norms. As part of these case studies, we collect requirements for exploratory, omniscient, multiverse debuggers, forming the basis of our evaluation. Concretely, we make the following contributions.

- We (re-)define the original framework by Pasquier et al. using set-theoretic notation (in Section 2).
- We investigate the application of exploratory, omniscient, multiverse debugging in three domain-specific languages, resulting in a set of user stories and requirements for each of the domains (in Sections 3 to 5).
- We evaluate the applicability of the framework against the requirements, providing further evidence in support of the use of a generic framework, whilst identifying a number of limitations (in Section 7).
- We introduce an extended version of the framework that (partially) addresses these limitations (in Section 6).

Sections 8 to 10 discuss the threats to validity of our work, related work and conclude (respectively).

2 The Original Multiverse Framework

Our work builds on the reusable multiverse debugging framework introduced by Pasquier et al. [20]. For consistency reasons, we define the framework using set-theoretic notation as an alternative to the Lean [8] code given in the original paper.

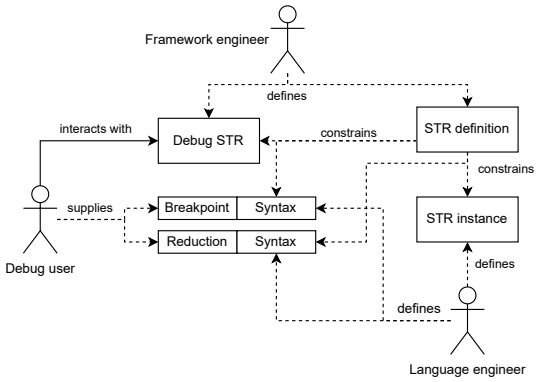

Figure 1. The relations between roles and components of the multiverse debugging framework.

An overview of the debugging framework and the relations between different roles is displayed in Figure 1.

Definition 2.1. A Semantic Transition Relation (STR) is a tuple $\langle C, C_0, A, I, Act \rangle$, where C is a set of configurations, $C_0 \subseteq C$ a set of initial configurations, A is a set of actions, $I : C \times A \to \mathcal{P}(C)$ is a non-deterministic interpreter of actions upon configurations, and $Act : C \to \mathcal{P}(A)$ determines the set of executable actions for a given configuration.

There are two contributors to non-determinism in an STR: (1) for every configuration, there might be multiple executable actions (Act) and (2) for every action acting upon a configuration, there might be multiple result configurations (I). The two contributors can be seen as stemming from a non-deterministic choice *internal* to the interpreter and an *external* non-deterministic choice.

Let $S = \langle C_s, C_{s0}, A_s, I_d, Act_s \rangle$ be a language STR, a multiverse debugger is defined in terms of this STR as follows:

$$D_S(R, B) = \langle C_d, C_{d0}, A_d, I_d, Act_d \rangle.$$

Parameter $R : C_s \to C_s/r$ is a reducer function reducing configurations to a reduced form C_s/r which support equality between elements. Parameter $B : C_s \to \mathbb{B}$ represents a breakpoint as a predicate over configurations, determining the configurations in which the breakpoint is 'activated'.

The configuration of a debugger is defined as a tuple: $C_d = \langle C_s \cup \{\bot\}, \mathcal{P}(C_s), \mathcal{P}(C_s) \rangle$. The first component denotes the current, object language-specific configuration or is $\bot$ when there is none. The second component is a history represented as a set of (previously) encountered configurations. The last component is a set of options to choose from after the (non-deterministic) execution of an action.

The debugger actions, set of available actions, and the interpreter are defined as follows. The interpreter is defined

$$\frac{a_s \in Act_s(c_s) \qquad c_s \neq \bot \qquad opts = I_s(c_s, a_s)}{\langle c_s, hist, \_\rangle \xrightarrow{step\ a_s} \langle \bot, hist, opts\rangle} \text{(STEP)}$$

$$\frac{c_s \in opts}{\langle \_, hist, opts\rangle \xrightarrow{select\ c_s} \langle c_s, \{c_s\} \cup hist, \emptyset\rangle} \text{(SELECT)}$$

$$\frac{c_s \in hist}{\langle \_, hist, \_\rangle \xrightarrow{jump\ c_s} \langle c_s, \{c_s\} \cup hist, \emptyset\rangle} \text{(JUMP)}$$

$$\frac{c_s \neq \bot \qquad find_{(R,B)}(\{c_s\}) = (c_{s1}, \ldots, c_{sn})}{\langle c_s, hist, \_\rangle \xrightarrow{run\_to\_breakpoint} \langle c_{s1}, hist \cup \{c_{s1}, \ldots, c_{sn}\}, \emptyset\rangle} \text{(RUN)}$$

$$\frac{find_{(R,B)}(opts) = (c_{s1}, \ldots, c_{sn})}{\langle \bot, hist, opts\rangle \xrightarrow{run\_to\_breakpoint} \langle c_{s1}, hist \cup \{c_{s1}, \ldots, c_{sn}\}, \emptyset\rangle} \text{(RUN-2)}$$

Figure 2. Semantics of the debugging operations. Under scores denote unused meta-variables, and can be replaced by an appropriate meta-variable as long as every underscore gets assigned a unique meta-variable. The subscript s denotes components of the underlying language STR.

in terms of a transition relation $\xrightarrow{a}$, defined by the inference system in Figure 2.

$$A_d ::= step\ A_s \mid select\ C_s \mid jump\ C_s \mid run\_to\_breakpoint$$
$$Act_d(\langle c_s, h, o\rangle) = \{step\ a_s \mid a_s \in Act_s(c_s), c_s \neq \bot\}$$
$$\cup \{jump\ c \mid c \in h\} \cup \{select\ c \mid c \in o\}$$
$$\cup \{run\_to\_breakpoint\}$$
$$I_d(c, a) = \{c' \mid c \xrightarrow{a} c'\}$$

For any (stepwise) interpreter I, we define the reachability graph as embedding all the possible execution traces from a given configuration. The definition is adapted from [29].

Definition 2.2. Let I_a be an interpreter for actions $a \in A$ and configurations $c \in C$. The *reachability graph* from a configuration $c \in C$ is the graph $\langle V, E\rangle$ with V and E the smallest sets of nodes and labelled edges such that $c \in V$ and for every triple $\langle c_1, a, c_2\rangle$, with $c_1 \in V$ and $c_2 = I_a(c_1)$, it holds that $c_2 \in V$ and that $\langle c_1, a, c_2\rangle \in E$.

The semantics of *run_to_breakpoint* is defined in terms of *find* (not defined here) performing a depth-first search in the reachability graph to find a configuration for which the B predicate succeeds. Throughout this search, a set of reduced configurations is maintained, containing the reduced versions of the configurations encountered during search by applying reduction function R. When a reduced configuration is revisited, the current search-branch terminates and *find* backtracks. If a configuration satisfying B is found, this configuration and its predecessors are returned as a sequence. If the search is exhausted, the empty sequence is returned. The reductions can yield a finite exploration of an infinite reachability graph. However, the algorithm does not terminate in the general case. See [20] for a more formal definition of *find*.

The functions R and B are the result of (partially) evaluating a breakpoint and reduction expression given by the programmer. The syntax for breakpoint- and reduction-expressions is determined by the language engineer (see Figure 1). The implementation of find in the original framework [20], described above, focused on breakpoints as predicates over configurations. The implementation needs to be modified for more expressive breakpoints. The implementation of *find* in [21] adds support for breakpoints over transitions and sequences of transitions with regular expressions and LTL-formulae as example formalisms.

3 Grammar and parser engineering

In this and the following two sections we present case studies across three different domains: grammar engineering, formal operational semantics, and norm engineering. For every domain we investigate usage scenarios in which a user attempts to locate, understand, and consider resolutions for (together: *diagnose*) a particular error, bug, or otherwise unwanted result. To this end, we describe each domain, define the most important user roles, and associate one or more user stories which each of the roles. The user stories are re-formulated as functional requirements for debuggers and, by extension, for an underlying debugging framework. The requirements are derived from the needs of (hypothetical) users of domain-specific debugger implementations and are used in Section 7 to evaluate the multiverse debugging frameworks discussed in this paper. The first case study investigates diagnosis in the context of grammar and parser engineering.

A context-free grammar (simply 'grammar', hereafter) specifies the concrete syntax of a (software) language. The conventional definition of a grammar, originally provided by Chomsky [5], associates one or more production rules with nonterminal symbols. A production rule consists of a sequence of nonterminal symbols and terminal symbols, with terminal symbols capturing the tokens (words) of a language. A nonterminal in a grammar derives sentences (sequences of tokens) through the recursive process of in-place replacing nonterminal symbols – with one of the productions associated with that nonterminal – until a sequence consisting of only terminal symbols is obtained.

This process is non-deterministic when at least one of the encountered nonterminal symbols has two or more associated production rules. As a result, the same nonterminal can be used to generate multiple sentences. Conversely, the same

sentence can be the result of alternative sequences of derivation steps starting from the same nonterminal. Grammars that have one or more such sentences[1] are *ambiguous*. The possibly many combinations of non-deterministic choices in the derivation process is the source of the great expressiveness of grammars, but also of (any) complexity in parsers.

A parser is an algorithm that attempts to determine whether a given input sentence can be derived from a nominated nonterminal symbol (the 'start symbol'). The evidence of a successful parse can be a parse tree effectively encoding the steps of a derivation process. For a more extensive take on grammars and parsing, the reader is referred to [1, 11].

In the context of software languages, ambiguities in a grammar are often considered as flaws of the grammar introduced by the *grammar engineer* who wrote the grammar[2]. Many examples of ambiguities in real-world software language definitions exist [32], e.g., in ANSI-C, and ambiguities can be notoriously difficult to detect [3]. An ambiguity can be resolved by a refactoring of the grammar that preserves the set of sentences generated by the grammar. Additional refactorings, such as left-factoring, can be performed to reduce the non-determinism of the grammar. Such refactorings are often performed, explicitly or implicitly, by a *parser engineer* responsible for implementing a parser for the language. The implemented parser should be sound with respect to the (original) grammar definition such that it accepts only derivable sentences. This is especially the case when the grammar forms a contract between the grammar engineer and the *programmer*, e.g., when the grammar is part of a reference manual. In this case, helpful errors messages provided by a parser can refer to the grammar to help solve syntax errors in a program. Figure 3 visualises the user roles, artefacts and relations in this (idealised) view on syntax and parsing.

Based on this view, we have defined four user stories for the various user roles. 1): As a grammar engineer, I want to diagnose and remove ambiguities in the grammar. 2): As a grammar engineer, I want to explore the set of sentences (the language) generated by the grammar. 3): As a parser engineer, I want to discover whether the grammar is in the class LL(1), as this enables (hand- written) deterministic recursive descent parsers. 4): As a programmer, I want to better understand syntax errors reported to me by a (deterministic or non- deterministic) top-down parser.

In the next subsection, we will use the multiverse debugging framework to define a debugger targeting these user stories. The STR of this debugger captures the derivation process as the described at the start of this subsection. Top-down and recursive descent parsers (such as LL(k) and GLL)

[1]Strictly speaking, a grammar is ambiguous iff there are multiple *left-most* or *right-most* derivations of a sentence.

[2]Although the developers of software language workbenches typically accept ambiguity as a consequence of a more 'natural' definition of a language and introduce ambiguity reduction annotations to the grammar formalism provided by the workbench to disambiguate under-the-hood.

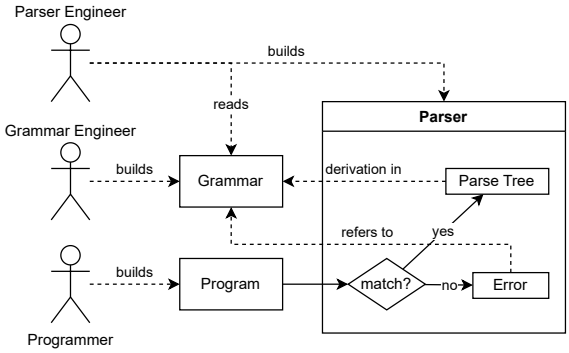

Figure 3. The ideal relation between the different roles and artefacts involved in the production and usage of grammars and parsers. The program is the input sentence to the parser.

implement an algorithm that resembles this process quite closely. As a result, a debugger that simultaneously aids engineers of grammars and top-down parsers is more likely (to be feasible and intuitive) than a debugger that combines diagnosis for grammars and bottom-up parsers (such as LR(k) and LALR). We have left bottom-up parsing out of scope in this paper.

3.1 An STR for grammars

The STR interface for debugging grammars is defined as follows. A configuration is defined as a tuple $\langle S, i \rangle$, where S is a stack and $i \geq 0$ is an index into some input sentence I. The input sentence I and the grammar $\mathfrak{G}$ do not occur in a configuration as they are constant per derivation/parsing session. The configuration $\langle (\mathcal{X} \rightarrow \cdot \mathfrak{G}_S), 0 \rangle$ is the single starting configuration in which $\mathfrak{G}_S$ denotes the nominated start symbol of the grammar and $\mathcal{X}$ a fresh, auxiliary start symbol used to detect a successful derivation (see Rule ACCEPT in Figure 4). The elements of the stack are quadruples displayed as $(X \rightarrow \alpha \cdot \beta, i)$, with i an index, $(X, \alpha\beta)$ a production, α, β are sequences of symbols, and $\cdot$ denoting the progress made in matching the production, with the symbols α already matched. Notationally, the stack is a sequence of elements separated by the $\bullet$ symbol, with the more recently pushed element on the right. We refer to the symbol after the $\cdot$ in the top-most element of the stack as the next symbol to match.

The actions are given by the following grammar:

$$a \in \mathit{actions} ::= \mathit{match}(t) \mid \mathit{descend}(X, \alpha) \mid \mathit{ascend} \mid \mathit{accept}$$

Meta-variable t refers to a token, X to a non-terminal symbol, and α to a sequence of symbols. The semantics of the actions are given as inference rules in Figure 4. The $\mathit{match}(t)$ action is available when terminal t is the next symbol to match and when the current index i points to t in the input sentence. The $\mathit{descend}(X, \alpha)$ action is available when nonterminal X is the next symbol to match and when $X \rightarrow \alpha$ is a production in the grammar. The descend action corresponds to the function call of a recursive descent parser in which

$$\frac{I_i = t}{\langle S \bullet (X \to \alpha \cdot t\beta, j), i\rangle \xrightarrow{match(t)} \langle S \bullet (X \to \alpha t \cdot \beta, j), i+1\rangle} \text{(MATCH)}$$

$$\frac{S = S' \bullet (Y \to \alpha \cdot X\beta, j) \quad (X, \delta) \in \mathfrak{G}}{\langle S, i\rangle \xrightarrow{descend(X,\delta)} \langle S \bullet (X \to \cdot\delta, i), i\rangle} \text{(DESCEND)}$$

$$\frac{S' = S \bullet (Y \to \alpha X \cdot \beta, k)}{\langle S \bullet (Y \to \alpha \cdot X\beta, k) \bullet (X \to \gamma\cdot, j), i\rangle \xrightarrow{ascend} \langle S', i\rangle} \text{(ASCEND)}$$

$$\frac{I_i = \$ \qquad S = (\mathcal{X} \to \mathfrak{G}_S\cdot, 0)}{\langle S, i\rangle \xrightarrow{accept} \langle S, i\rangle} \text{(ACCEPT)} \qquad I(c, a) = \{c' \mid c \overset{a}{\Rightarrow} c'\} \text{ (Interpreter)}$$

Figure 4. Semantics of the grammar-engineering domain-specific debug actions, and the resulting interpreter. For simplicity, the rules encode LL(1). LL(k) can be obtained with slight modifications to the handling of indices in the rules.

the stack resembles the call-stack of the parser. The *ascend* action is available when there is no next symbol to match and is analogous to returning from a call to a recursive descent parser. The *accept* action is available only when the stack indicates the start symbol of the grammar has been matched and the end of the input has been reached, indicating a completed derivation/parsing process. The naming of the actions is inspired by the characterisation of recursive descent (top-down) parsing algorithms given in [27].

3.2 User interactions

A debugger is considered to satisfy a user story if it affords a sequence of user interactions that together (sufficiently) support the user in realising the goal formulated in the user story. In the analysis we focus on a theoretical/objective realisation of the story rather than user experience. That is, we determine whether the sequence of interactions yields the informational content needed for the diagnosis, not on how this information is made available or presented.

To explore suitable interactions for the ambiguity user story, let i, j be integers, and X a nonterminal. All configurations of the form $\langle S \bullet (X \to \gamma\cdot, j), i\rangle$ in the reachability graph generated by the interpreter indicate that the nonterminal X can derive the subsentence $I_{j,i}$ of the input I ranging from j to i. The grammar engineer can inspect the reachability graph in two ways: (1) choosing a concrete γ and finding multiple paths in the reachability graph reaching the corresponding configuration and (2) finding multiple configurations for different choices of γ. In both cases, known as horizontal and vertical ambiguity respectively [3], there is a (left-most) derivation of $I_{j,i}$ per path and the path gives the specific and concrete details of the derivation. Analysing and comparing these paths gives insight into the nature of the ambiguity, which can then be used to resolve the ambiguity by modifying the grammar. A feasible reduction discards the tail of the stack (S above) from a configuration. Figure 8 gives several examples of breakpoint and reduction expression for this case study. Based on these observations, we define the following two requirements.

Requirement 1 (RQ1)

A debug user should be able to define breakpoints over paths.

Requirement 2 (RQ2)

A debug user should be able to find all configurations satisfying a breakpoint

The first requirement supports the first case of ambiguity, where a user can use a breakpoint to find a configuration with multiple incoming edges. The second requirement supports the second case of ambiguity, where a user can find all configurations that project the same information (X, i and j in this case).

A grammar is left recursive when it contains a production rule $X \to \alpha$ for which holds that the derivation process applied to α can yield a sentence of the form $X\gamma$, for some γ, via one or more derivation steps. The reachability graph generated by the interpreter applied to a left-recursive grammar is infinite as infinitely many descend actions on X can be performed whilst ever-growing the stack. To still utilize breakpoint finding in such scenarios, a bounded search can be applied. Hence, we formulate the following requirement. Instead of indiscriminately bounding the search space, we can also filter out only those paths where the stack displays more 'recursive calls' than elements left in the input sentence (an approach to handle left-recursion suggested in [10]). We therefore formulate the following requirement. Based on these observations, we formulate the following two requirements.

Requirement 3 (RQ3)

A debug user should be able to control the depth of breakpoint searches.

Requirement 4 (RQ4)

A debug user should be able to pre-emptively reduce (prune) the search space.

The second user story is related to sentence generation. The STR can be used to generate sentences by (selectively) ignoring I when determining whether *match* and *accept* actions are available. Paths in the reachability graph ending with an *accept* transition then display sentences derivable from the start symbol (by inspecting the *match*-transitions of the path). Note that there may be infinitely many such paths. A grammar engineer might be interested in sentences of a particular structure by selectively disabling the input sentence. The order of the sentences generated via *find* is determined by the implemented search strategy. And a depth-first search strategy may not yield a representative sample when applied a limited amount of times.

Requirement 5 (RQ5)

A debug user should be able to control the search strategy used during breakpoint finding.

The third user story is related to the implementation of the parser. If the grammar is in the class of LL(1) grammars, the parser engineer can hand-write a performant recursive descent parser. We modify the debugger by adding a condition that makes an action $descend(X, \alpha)$ available only if I_i is in the *first-set* of α (first-sets can be precomputed from the grammar definition [11]). A concrete counter-example to the LL(1) property is found if there is a configuration admitting two or more *descend* actions. Requirements 1, 2, 3 suffice.

The fourth and final user story is related to a programmer interacting with a parser. The programmer wants to better understand the error they made in a program rejected by a parser. The programmer can use the debugger by finding configurations not admitting any actions (the derivation cannot continue and is not complete). The programmer can then inspect the trace of the current execution to obtain information regarding the parse error. A top-down parser typically performs LL(1)-lookahead. Thus we need the first-set condition described above. If the parser is deterministic, and the grammar LL(1), then our debugger will run into the same unique error. If the parser is non-deterministic, we can use the debugger to find all error states discovered by the parser. Basic breakpoints and Requirement 2 suffice.

4 Funcons

Defining the semantics of a programming language is complex, but programming languages often share concepts and constructs, giving reuse opportunities. In this case study we explore debugging in the context of the funcons framework for defining the operational semantics of programming languages with reusable components referred to as 'funcons' [17]. The framework provides a library of *fun*damental *con*structs (*funcons*) as building blocks for the operational semantics of programming languages. Every funcon is formally defined by a *funcon engineer* in the CBS meta-language [17],

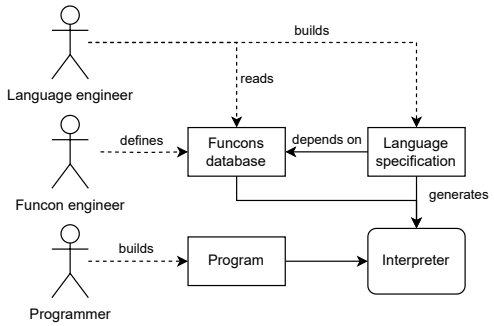

Figure 5. The ideal relation between the different roles and artefacts involved in the production and usage of operational semantics and (derived) interpreters.

using small-step I-MSOS [16, 18], a modular variant of structural operational semantics (SOS) [22]. The operational semantics of a programming language is defined by a *language engineer* via a translation from object language programs to funcon terms. A *programmer* (language user) writes a program in the object language as input to an interpreter that first converts the input program to a funcon term (following the translational semantics) and then applies the existing funcon term interpreter to the resulting funcon term [30]. In this case study we investigate whether and how the multiverse debugging framework can be used to obtain a multiverse debugger for both funcon terms and object language programs. Figure 5 visualises the aforementioned user roles, artefacts and relations in this (idealised) view on reusable programming language semantics.

The funcon term interpreter is directly derived from the small-step definition of the funcons written in CBS [30]. The small-step nature of these definitions makes it possible to instrument the funcon interpreter to enable stepwise debugging, with each step in the underlying SOS semantics corresponding to a step in the debugger[3]. The funcon interpreter is non-deterministic if at least one of the funcons in the funcon library is non-deterministic, which is the case if: The funcon is defined by (I-MSOS) rules that are not mutually exclusive, i.e., two or more rule instances[4] can be simultaneously applied to perform a step on a given funcon term. Or, the funcon term is given an informal or axiomatic definition that is inherently non-deterministic. The first source of non-determinism is generally considered to be undesirable when defining the semantics of a deterministic or sequential programming language. However, one can use this source of non-determinism to specify the behaviour of concurrent programming constructs (e.g., threads) and non-deterministic

[3]Note that this process may not result in the desired granularity of steps.
[4]In SOS and variants, a rule is instantiated to form a rule instance by substituting meta-variables, not all of which may be bound by the term under evaluation, creating a source of non-determinism.

operators (e.g., without a well-defined order of argument evaluation).

In the current funcon library, `set-elements` is a non-deterministic operator which returns a permutation of the elements of a set. Crucially, the order of the returned sequence is unspecified. This funcon is directly or indirectly used in the definition of other non-deterministic funcons such as `some-element` (yielding an arbitrary element from a non-empty set), which is used to define the semantics of concurrent programming languages based on the thread-model.

Based on the aforementioned described domain, we have defined five user stories. 1): As a funcons engineer, I want to introduce new (non-deterministic) funcons and explore their semantics and interaction with existing funcons. 2): As a language engineer, I want to experiment with funcons to determine the right combination of funcons for my language semantics. 3): As a language user, I want to query the current program state. 4): As a language user, I want to see the code around the current program point. 5): As a language user, I want to modify the program and observe the effects of the modification.

4.1 A STR for funcons

The STR configuration for funcons consists out of the current (funcon) term under execution and a set of auxiliary semantic entities capturing contextual information such as variable bindings, assignments and printed output (see [30]). The funcon debugger has one action: `step`. Its semantics and the resulting interpreter are as follows.

$$\frac{t \xrightarrow{(e,e')} t'}{\langle t, e\rangle \xRightarrow{step} \langle t', e'\rangle}\ (\textsc{step}) \qquad I(c,a) = \{c' \mid c \overset{a}{\Rightarrow} c'\}\ (\text{Interpreter})$$

The `step` action steps the current term and updates the configuration with the derived term and the updated entities, if any. If the step triggers the evaluation of a non-deterministic funcon, the step may yield more than one output configuration.

4.2 User interactions

The first user story is from a funcon engineers viewpoint who wants to explore the right definition for a non-deterministic funcon. A funcon is non-deterministic if the reachability graph contains a configuration with multiple outgoing edges. Such configurations can be easily found using Requirement 1. However, even with a small-step semantics, one step can introduce many new configurations. This can happen either when the root term directly produces many result configurations, or because sub-computations produce many result configurations, which can aggregate. So, performing one step can still cause an enormous growth of the state space, such that it is infeasible to control manually. Nevertheless, not all the non-determinism observed during a step is of interest. Therefore, by focusing only on non-determinism that is significant to the debugging task at hand, the amount of states produced by one step can be significantly reduced. We therefore formulate the following requirement.

Requirement 6 (RQ6)

A debug user should be able to control for which non-deterministic terms all states are visited.

The second user story is from the perspective of a language engineer who uses funcons to give semantics to an object language. This type of user might utilize the debugger to explore semantics for a specific language construct by giving a definition for that construct in terms of funcons and testing the construct. The language engineer can achieve this by starting a new debugging session for every example program and observing the behaviour. However, that makes it difficult to compare debug sessions, which is a primary goal in this user story. With that in mind, we define the following requirement.

Requirement 7 (RQ7)

A debug user should be able to go back-in-time to retry a scenario with different input values and compare the outcome values of the different scenarios.

The third and fourth user stories are from the perspective of a programmer using an object-language with semantics defined in terms of funcons. Both these user stories describe interactions with the debugger that enable a programmer to get a better idea of the current state the program is in, without getting overloaded with too much information. The third user story does this by enabling the user to query specific information out of a specific state. And the fourth user story enables a user to build a better mental model where the execution is paused. Both of these interactions can be achieved via the breakpoint and reduction functionality already present in the debugger. A query on the state can be formulated as a breakpoint, if the answer is boolean. Alternatively, a reduction and display of the reduced configuration can project specific information out of configurations. We therefore formulate the following requirement.

Requirement 8 (RQ8)

A debug user should be able to test for breakpoints and to visualize (reduced) configurations.

5 eFLINT (reasoning with norms)

The eFLINT language is a domain-specific language for reasoning with formalized interpretations of norms as they are found in laws, regulations, contracts and organisational policies [28]. In eFLINT, a normative specification encodes

a formal interpretation of norms, declares data-structures (types) for knowledge representation whose instances (facts) are either true or false. This part of a specification is referred to as the *ontology* of the specification. In the *process model* of a specification, effects are associated with action- and event-types, determining which facts are rendered true or false by the triggering of instances of these types (actions and events, respectively). Together, the ontology and the process model describe a finite, non-deterministic state automaton in which states are formed by the set of (true) facts and transitions are formed by the (effects of) actions and events. The automaton is non-deterministic in that in any given state, multiple actions and/or events may be triggereable. The automaton is finite as there is a finite amount of (possible true) facts and because the amount of outgoing transitions of any state is bounded by the finite set of possible actions and events.

The *normative classification*, the final part of an eFLINT specification, assigns violations to states and transitions of the automaton[5]. A duty-type declaration establishes that a state is violating if it states the truth of an instance of the type (a duty) whilst also satisfying one or more of the violation conditions (Boolean expressions over facts) associated with the duty-type. An action-type declaration establishes that a transition is violating if one or more of the pre-conditions (Boolean expressions over facts) associated with the action-type is not satisfied by the source-state of the transition.

The language can be used to establish the extent to which a software system complies with (the formalised interpretation of norms encoded in) a policy document. In an idealised setting, visualised in Figure 6, a *policy expert* determines the policy – possibly including relevant laws and regulations – of an organisation employing some software system. Following a model-driven approach, a *software engineer* maintains[6] both the running software system as well as the parts of the eFLINT specification that model the software system (ontology and process model). The *policy engineer* extends this eFLINT specification with the normative classification, established by formalizing the norms encoded in the policy[7].

Based on the above description of the domain, we formulate the following five user stories. 1): As a policy engineer, I want to discover in what ways particular transitions or states can be reached. 2): As a policy engineer, I want to discover the effects on possible violations of certain modifications to the normative classification of an eFLINT specification. 3): As a policy engineer, I want discover how to modify the normative classification to ensure certain states or transitions are (no longer) violating. 4): As a software engineer, I want to assess the compliance risks of the software system modelled by an eFLINT specification by determining in what ways violating states and transitions can be reached. 5): As a software engineer, I want to modify the process model of an eFLINT specification to reduce the number of possible occurrences of violations.

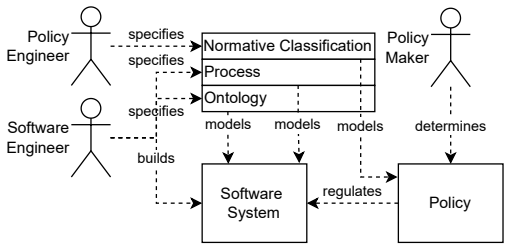

Figure 6. The ideal relation between the different roles and artefacts involved in checking the compliance of a software system against policy using eFLINT.

5.1 A STR for eFLINT

The eFLINT STR consists of a configuration defined as a tuple containing the current specification and the current knowledge base. eFLINT has one type of action: *trigger t*. A *trigger* action is generated for every trigger-able action in the knowledge base. The semantics of the debugging action and the resulting interpreter are as follows.

$$\frac{t \in kb \qquad kb \xrightarrow{t} kb'}{kb \xrightarrow{\text{trigger } t} kb'} \text{(TRIGGER)} \qquad I(c, a) = \{c' \mid c \overset{a}{\Rightarrow} c'\} \text{ (Interpreter)}$$

5.2 User interactions

For the first user story, the user wants to find states that are reached in a particular way. This user story is already captured by Requirement 1.

The second user story is focused on comparing paths or configurations at different points in a debugging session. We therefore formulate the following requirement.

Requirement 9 (RQ9)

A debug user should be able to inspect partial debug traces.

This requirement differs from Requirement 7 in two aspects: no back-in-time functionality is required, and only one particular trace is inspected.

The third user story is concerned with assessing the effects of modifications to the normative specification, such as the possible violations that can occur in a system. This can be achieved by altering the specification being debugged and

[5] In practice, a normative classification will also introduce sets of *institutional* facts and actions that play a role in establishing compliance, separate from the *physical* facts and actions representing the software system, see [19, 25, 28, 31].

[6] We are not concerned here with whether or how one is derived from the other or how the two are kept consistent.

[7] We are also not concerned with the processes required to integrate concepts from the policy and software system

re-evaluating a particular scenario. The requirement for such interactions correspond to Requirement 7.

The fourth user story concerns finding compliance related breakpoints for a scenario in a specification. A user could achieve this user story by defining a breakpoint that finds states in which a violation exists. To find multiple such points, the debugger needs to support finding multiple breakpoints, as formulated in Requirement 2. To also determine how these different breakpoints were reached, the debugger needs to keep track of multiple histories and make multiple histories insightful. Hence, we formulate the following requirement.

Requirement 10 (RQ10)

A debug users should be able to operate on multiple histories in one debug session.

The fifth user story concerns the viewpoint of a systems engineer that wants to reduce violations in their system by modifying the process. This user story is similar to the third user story, but from the viewpoint of a different actor. Nevertheless, the specific user interaction is the same. Hence, the requirements needed to satisfy the current user interactions are also captured by Requirement 7.

6 Generalized Multiverse Debugging

We introduce the generalized STR (GSTR) which adapts the STR-based debugging framework along three dimensions: new components for meta-actions are added to the STR, the history and options component are generalized using graph structures instead of sets, and the breakpoint (B) and reduce (R) functions are generalized by *Step* and *Label* functions.

The meta-action components are motivated by Requirements 7,8, and the removal of the breakpoint and reduce functions. Using meta-actions, language engineers can extend their debugger with language-specific functionality. This is extra useful for visualisation and query operations which do not alter the configuration in any way, but as actions would still be included in the history. With meta-actions, the functionality remains without polluting the history.

Using graphs instead of sets for the history and options component is motivated by Requirements 1,9. With graphs, more information is retained, empowering more expressive breakpoint and reduction functions.

The last adaptation generalizes the breakpoint and reduce functions by *Step* and *Label* functions, and is motivated by Requirements 2,3,4,5. The *Label* function assigns a label to every configuration in the current search graph. The *Step* function performs a step on the current search graph based on the labelling by the *Label* function. After a step, the search graph is extended with new configurations and another iteration of labelling and stepping is performed. When no new configurations are added the algorithm stops. Compared to the original breakpoint (B) and reduce (R) functions, the *Step* and *Label* functions provide more flexibility and expressiveness, while also promoting reusability among different language-specific debuggers. For instance, in our implementation we have defined several reusable functions that can be combined to create concrete *Step* and *Label* instances. It is thus possible to implement new search strategies without modifications to the debugging framework.

6.1 Formal generalized-STR definition

We now give the formal definition of the GSTR, following the formal definition in Section 2.

Definition 6.1. A generalized STR (GSTR) is a tuple $\langle C, C_0, A, M, I, Act, P, Com\rangle$, which extends the STR with three new elements: M, P, Com. M denotes a set of meta-actions (or commands). P is a function $C \times M \to C \times O$ which performs a command on a configuration, resulting in an updated configuration and an output value. The set O is left abstract but is defined by the debugging framework and varies depending on the execution environment. It provides a mechanism for meta-actions to communicate with the external world.[8] Finally, the Com component is a function $C \to \mathcal{P}(M)$ giving the active commands for the given configuration.

A generalized debugger is defined in terms of a GSTR, where $GS = \langle C_s, C_{s0}, A_s, M_s, I_s, Act_s, P_s, Com_s\rangle$ is the GSTR for the language being debugged:

$$GD_{GS}(Step, Label) = \langle C_d, C_{d0}, A_d, M_d, I_d, Act_d, P_d, Com_d\rangle.$$

The debugger configuration is defined as a tuple: $C_d = \langle C_s \cup \{\bot\}, \mathcal{G}(C_s, A_s), \mathcal{G}(C_s, A_s)\rangle$, with $\mathcal{G}(C, A) = \langle \mathcal{P}(C), \mathcal{P}(C \times A \times C)\rangle$ denoting a graph with vertices being elements of C and edges are labelled by elements of A. The set of actions A_d is the set of actions of STR debugger extended with a *meta m* action for execution of the meta-commands in M_s. For the debugger, we leave the set of meta-commands (M_d) empty. Therefore, the function P_d is a constant function returning the given configuration and no output. The debugger is indexed by two functions: *Step* and *Label*, which generalize over the *Breakpoint*(B) and *Reduce*(R) functions from the STR definition, which is discussed in more detail in Section 6.2.

$$Step : (C \to L) \to \mathcal{G}(C, A) \to \mathcal{G}(C, A)$$
$$Label : \mathcal{G}(C, A) \to (C \to L)$$

The set L is a label set with elements forming labels. Every label set comes associated with two functions $\langle accept, enabled\rangle$ of type $L \to \mathbb{B}$. The *accept* function denotes whether a particular label indicates that the associated configuration is an accepting state. The *enabled* function denotes whether a particular label indicates that the associated configuration is enabled for transitions. The *Step* function iterates the graph in such a way that only new outgoing edges are added to

[8] An alternative method to model external communication is to execute the meta-action in a monad. For simplicity, we have opted to model it using an abstract output value.

$$\frac{a_s \in Act_s(c_s) \qquad c_s \neq \bot \qquad cs = I_s(c_s, a_s) \qquad opts = (cs, \{(c, a, c') \mid c' \in cs\})}{\langle c_s, hist, \_\rangle \xrightarrow{step\ a_s} \langle \bot, hist, opts\rangle} \text{(STEP)}$$

$$\frac{c_s \in \mathcal{V}(opts) \qquad g' = g \cup_G opts}{\langle \_, g, opts\rangle \xrightarrow{select\ c_s} \langle c_s, g', \epsilon\rangle} \text{(SELECT)}$$

$$\frac{c_s \in \mathcal{V}(g)}{\langle \_, g, \_\rangle \xrightarrow{jump\ c_s} \langle c_s, g, \epsilon\rangle} \text{(JUMP)}$$

$$\frac{c_s \neq \bot \qquad m_s \in Com_s(c_s) \qquad M_s(m_s) = \langle c', o\rangle}{\langle c_s, g, \epsilon\rangle \xrightarrow{meta\ m} \langle c', g \frown c', \epsilon\rangle} \text{(META)}$$

$$\frac{find_\psi((\{c_s\}, \emptyset)) = (gn, cn)}{\langle c_s, g, \_\rangle \xrightarrow{run\_to\_breakpoint} \langle \bot, g \cup_G gn, cn\rangle} \text{(RUN)}$$

$$\frac{find_\psi(opts) = (gn, cn)}{\langle \bot, g, opts\rangle \xrightarrow{run\_to\_breakpoint} \langle \bot, g \cup_G gn, cn\rangle} \text{(RUN-2)}$$

$$I_d(c_d, a_d) = \{c'_d \mid c_d \xrightarrow{a} c'_d\} \qquad \text{(Interpreter)}$$

Figure 7. Semantics of the debugging operations for the generalized debugger. The function $\mathcal{V}$ projects the vertices out of a graph. The $\frown$ operation adds a configuration to the vertices of a graph. The $\cup_g$ operation combines two graphs by taking the pointwise union. We use ϵ for empty graphs.

vertices with a label marked as *enabled*. The debugger itself uses the *accept* function to extract the *interesting* configurations after a search. The iterative process is performed until a fixed-point is reached, which requires that the *Step* function is monotonic on the structure of the graph. Finally, the semantics of the debugger (I_d) is updated, shown in Figure 7.

Compared to the STR definition, the GSTR definition does not use *Break* and *Reduce* functions. Instead, *Step* and *Label* functions are used, and the history and options components are now generalized to graphs. This generalization is the biggest contributor to the changes required in the semantics of the debug actions.

6.2 Concrete *Step* and *Label* components

To show the expressiveness of the *Step* and *Label* functions, we highlight some of the breakpoint and reduction expressions from our grammar case study, and explain how to obtain the functionality of the original breakpoint and reduce functions using *Step* and *Label* functions.

Figure 8 highlights several breakpoint and reduction expressions defined for our grammar case study. Several of these examples were discussed from the user-interaction point-of-view in Section 3.2. The first breakpoint finds configurations denoting a successful parsing derivation. For this breakpoint, the *Label* function checks for every configuration if the condition is met, and if so the configuration is labelled as accepting. The second breakpoint finds configuration for which there exist two unique paths to some other configuration. The first reduction reduces configurations in which the next terminal to match is not a member of the computed follow-set. The second reduction reduces configurations where the stack is larger than the size of the input not yet matched. Both reductions do not work with a seen set, and instead prune, by labelling the configuration disabled, the search space immediately when a configuration that satisfies the expressions is found.

Breakpoints:

$$(\forall c)(\exists c')(c \xrightarrow{accept} c'). \qquad \text{(Accepting states)}$$

$$(\forall c)(\exists c', p, p')(c' \xrightarrow{p}_m c \wedge c' \xrightarrow{p'}_n c \wedge p \neq p'). \qquad \text{(Ambiguity points)}$$

Reductions:

$$(\forall c)(\mathcal{I}[c.i] \notin \mathcal{F}(c)).$$

$$(\forall c)(\exists X, \alpha)(count(\texttt{descend}(X, a), c.S) > length(\mathcal{I}) - c.i).$$

Figure 8. Example breakpoint and reductions applicable to the GSTR of the grammar-engineering case study. We use $\mathcal{F}(c)$ to denote the follow-set. Traces longer than 1 step are subscripted to denote the length of the trace.

To obtain the breakpoint/reduce functionality from the STR-debugger, we define a label set with three labels: *enabled*, *disabled*, and *accepted*. The *Label* function maps the graph to a reduced graph according to some reduction function, and assigns labels to the configurations according to the original semantics of the STR-based debugger: *accepted* if a configuration matches the breakpoint, *enabled* if a reduced configuration has no outgoing edges, and *disabled* otherwise. The *Step* function does a depth-first search until there exists a configuration with an *accepted* label or until there exists no configuration with an *enabled* label. The implementation is parametrized by the reduction and breakpoint function. This parametrization is fully hidden from the debugging framework.

7 Satisfaction of the Requirements

Table 1 discusses for each framework whether it satisfies a requirement and if not which modifications can be made to the framework to satisfy the requirement.

At a high level, the first five requirements are met by our framework due to the introduction of the *Step* and *Label* functions. Some of the requirements can be met by the original

framework via a small modification of the semantics, for example via an alternative implementation of the *find* function. This is also what we observed during the implementation of the case-study debuggers. Based on these observations, we came to the generalization via the *Step* and *Label* components that encompass all those requirements while also offering reusability and flexibility between different debugger implementations. With the *Step* and *Label* components, new search strategies, essentially alternative implementations for the *find* function, can be defined without needing modifications to the debugging framework semantics. Therefore, a *Language engineer* is not dependant on a *Framework engineer* when desiring alternative search strategies.

The second set of the requirements is more focused on the history mechanism, and most requirements are met by both frameworks. Still, the introduction of the graph-based history adds several new possibilities to the debuggers, including the generalization of the reduce and breakpoint functionality, while also supporting multiple independent debugging explorations in the same debugging session.

Finally, Requirement 6 is met by both frameworks, but not using a reusable mechanism. Instead, the required work is pushed to the interpreter of the language being debugged. Satisfying this requirement in a reusable manner requires interaction between the debugger and interpreter on every sub-computation, which requires severe alterations to the interpreter implementations. By not integrating this, we keep the framework interface simpler for languages that do not need the support for this feature, while still supporting the feature for languages that require it.

8 Threats to Validity

In this section we discuss the threats to validity present in this work from the point of view of empirical software research [6]. The primary component in our research is the requirements. The validity of the requirements can be affected by the chosen domains, and the selected user stories.

The selected user stories were determined by the authors based on expert experience in the respective domains. A more diverse set of user stories could be obtained by performing interviews with users across different experience levels. However, due to the fact that two out of the three domains have a small user base this was deemed impractical. In our case, we have thus opted to base our user stories on the experience of an expert in the respective domains.

To ensure our approach is transferable to different domains, we have performed our approach on three different domains. Furthermore, by being able to define the old framework in terms of the new framework, we retain the applicability of our work on those case studies.

9 Related work

The original multiverse debugging paper [15] presented Voyager, a multiverse debugger focused on AmbientTalk programs. As part of this implementation, they stored the exploration graph using the ArangoDB graph database. Hence, the graph of a debugging session can be queried using the ArangoDB query language. Our work is essentially a combination of the graph idea applied to the reusable framework of Pasquier et al. [21]. Although our implementation does not run on a (commercial) graph database, this is achievable in future work. Alternative options are also interesting, especially in combination with the *Step* and *Label* components. For example, using a graph algorithm language based on semigroups [13] or Kleene algebras [7] to guide the search. These ideas have been partially explored by [21] in the context of temporal breakpoints.

Omniscient debugging [14] enables back-in-time debugging, but does not have an explicit focus on non-deterministic programs. A reusable framework for omniscient debugging exists [4]. In addition, a significant amount of optimizations exist for omniscient debugging systems [2, 23]. In future work, it would be interesting to see if some of the optimizations can be applied to our framework, and how much work is required to extend existing omniscient debugging frameworks with multiverse debugging support.

With our debugger, a sequence of debugging interactions can be retried between different scenarios; a feature that is inspired by exploratory programming. Previous work [9] has done a deep-dive into the implications of different history mechanisms to store the exploration graph. Using a graph gives rise to all traces, which can encompass more traces than actively explored by the user. The full implications of this in the debugging context requires future work to determine. Nevertheless, our framework can be extended to keep a log of actions to reconstruct the concrete traces debugged by the user.

10 Conclusion

Debugging non-deterministic programs is challenging, primarily due to the state space explosion problem. Multiverse debugging aims to aid debugging non-deterministic programs by making debugging an interactive and user-controlled exploration of the state space. Previous work introduced a reusable framework for multiverse debugging, with the addition of user-definable reductions with which the search space can be reduced. Based on this framework, we have collected requirements for multiverse debuggers using three case studies in three different domains. Based on these requirements we have identified several limitations in previous work. Using these insights, we have introduced a modified and extended framework for multiverse debuggers with more general applicability while promoting reusability.

Table 1. Analysis of the extent to which the debugging framework of [20] and our extended version satisfy the requirements formulated for the case studies of this paper.

Requirement	Pasquier et al. [20]	Our work
RQ1	This requirement is met using a modified *find* function as demonstrated by [21].	This requirement is met owing to the introduction of the graph history and the *Label* function over this history, which can be defined such that it assigns a breakpoint label to configurations based on paths in the graph.
RQ2	This requirement can be met via a modification to the *find* function of the debugging framework, which currently performs a depth-first search and stops after finding a breakpoint.	This requirement is met owing to the definition of the *find* function in terms of the *Step* and *Label* functions. The *Step* and *Label* functions can be defined such that they continue the search until *all* breakpoints are reached. Termination of this process depends on the concrete *Label* function, and the underlying language being debugged.
RQ3	This requirement is not met due to the fixed semantics of the *find* function, which continues until either all (reduced) configurations have been visited or a breakpoint is reached. Nevertheless, this requirement is easily satisfied by modifying the *find* function to take an integer parameter to control the recursion depth of the search.	This requirement is met by defining a *Label* function that disables all configurations when there exists a non-repeating path in the graph of certain length. By disabling all configurations, the *Step* function cannot progress on any configuration, and the search will be terminated.
RQ4	This requirement can be met via a modification to the *find* function of the debugging framework, which currently utilizes a set of previously *seen* (reduced) configurations to handle pruning of the search space.	This requirement is met by defining a *Label* function that performs pruning based on properties of configurations in the graph. The previously *seen* (reduced) configurations is an example of such a property, but other implementations are possible, such as the pruning of left-recursion in the grammar case study.
RQ5	This requirement can be met by modifying the *find* function of the debugging framework, which currently performs a depth-first search.	This requirement is met by defining *Step* functions with different search strategies. For our case studies, we implemented depth- and breadth-first search. Other strategies, such as a parallel search strategy, are also possible.
RQ6	This requirement is met, but not in a reusable manner. Instead, the language designer needs to encode this functionality in the interpreter and use the configuration to communicate which non-deterministic terms need to be collapsed and which terms need to be fully explored.	This requirement is met, but not in a reusable manner. Instead, the language designer needs to encode this functionality in the interpreter and use the configuration to communicate which non-deterministic terms need to be collapsed and which terms need to be fully explored.
RQ7	This requirement is met via the support of user-definable actions and *jump*. A language engineer can add an action that modifies the program being debugged. A user can then *jump* to the specific configuration and perform the modification action at that point.	This requirement is met via the support of user-definable actions and *jump*. A language engineer can add an action that modifies the program being debugged. Meta-actions can be used instead, adding an isolated configuration to the history, resulting in a clearer divide between different explorations in the same debugging session.
RQ8	This requirement is not met by the framework because breakpoints and reductions are purely available inside the *find* function. Nevertheless, it would be trivial to extend the framework with this semantics by adding two new actions to the debugger, one to test a breakpoint on the current configuration, and one to reduce the current configuration.	This requirement is met by our framework via the usage of meta-actions in combination with the output result. Direct support from the framework for this requirement is thus removed. Requiring the usage of meta-actions to satisfy this requirement does move some implementation efforts away from the framework to the language engineer.
RQ9	This requirement can be satisfied through a relatively simple modification to the framework, using a tree or list to represent history instead (also discussed in [20]).	This requirement is satisfied owing to storing the history as a graph, which makes it trivial to focus on (partial) traces of the current debugging session.
RQ10	This requirement is satisfied via the *jump* action, which makes it possible to go back to a previous configuration and explore a different part of the history, thus supporting multiple histories in one debugging session.	This requirement is satisfied via the *jump* action, which makes it possible to go back to a previous configuration and explore a different part of the history, thus supporting multiple histories in one debugging session.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. I–X, 1–796 pages.

[2] Earl T. Barr and Mark Marron. 2014. Tardis: affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 67–82. https://doi.org/10.1145/2660193.2660209

[3] Hendrikus J. S. Basten and Jurgen J. Vinju. 2012. Parse Forest Diagnostics with Dr. Ambiguity. In *Software Language Engineering*, Anthony Sloane and Uwe Aßmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–302.

[4] Erwan Bousse, Dorian Leroy, Benoît Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient debugging for executable DSLs. *J. Syst. Softw.* 137 (2018), 261–288. https://doi.org/10.1016/J.JSS.2017.11.025

[5] N. Chomsky and M.P. Schützenberger. 1963. The Algebraic Theory of Context-Free Languages*. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 118–161. https://doi.org/10.1016/S0049-237X(08)72023-8

[6] Daniela S Cruzes and Lotfi ben Othmane. 2017. Threats to validity in empirical software security research. In *Empirical research for software security.* CRC Press, 275–300.

[7] Nikita Danilenko. 2015. *Designing Functional Implementations of Graph Algorithms (Entwurf funktionaler Implementierungen von Graphalgorithmen).* Ph. D. Dissertation. Kiel University, Germany. https://nbn-resolving.org/urn:nbn:de:gbv:8-diss-186649

[8] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[9] Damian Frölich and L. Thomas van Binsbergen. 2021. A Generic Back-End for Exploratory Programming. In *Trends in Functional Programming - 22nd International Symposium, TFP 2021, Virtual Event, February 17-19, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12834)*, Viktória Zsók and John Hughes (Eds.). Springer, 24–43. https://doi.org/10.1007/978-3-030-83978-9_2

[10] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-Recursive Grammars. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science, Vol. 4902).* Springer Berlin Heidelberg, 167–181. https://doi.org/10.1007/978-3-540-77442-6_12

[11] Dick Grune. 2010. *Parsing Techniques: A Practical Guide* (2nd ed.). Springer Publishing Company, Incorporated.

[12] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017*, Austin Z. Henley, Peter Rogers, and Anita Sarma (Eds.). IEEE Computer Society, 25–29. https://doi.org/10.1109/VLHCC.2017.8103446

[13] Donnacha Oisín Kidney and Nicolas Wu. 2025. Formalising Graph Algorithms with Coinduction. *Proc. ACM Program. Lang.* 9, POPL (2025), 1657–1686. https://doi.org/10.1145/3704892

[14] Bil Lewis. 2003. Debugging Backwards in Time. *CoRR* cs.SE/0310016 (2003). http://arxiv.org/abs/cs/0310016

[15] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. 2019. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:30. https://doi.org/10.4230/LIPICS.ECOOP.2019.27

[16] Peter D. Mosses. 2004. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 195–228.

[17] Peter D. Mosses. 2019. Software meta-language engineering and CBS. *Journal of Computer Languages* 50 (2019), 39–48. https://doi.org/10.1016/j.jvlc.2018.11.003

[18] Peter D. Mosses and Mark J. New. 2009. Implicit Propagation in Structural Operational Semantics. *Electronic Notes in Theoretical Computer Science* 229, 4 (2009).

[19] Pablo Noriega, Amit K. Chopra, Nicoletta Fornara, Henrique Lopes Cardoso, and Munindar P. Singh. 2013. Regulated MAS: Social Perspective. In *Normative Multi-Agent Systems*, Giulia Andrighetto, Guido Governatori, Pablo Noriega, and Leendert W. N. van der Torre (Eds.). Dagstuhl Follow-Ups, Vol. 4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 93–133. https://doi.org/10.4230/DFU.VOL4.12111.93

[20] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. 2022. Practical multiverse debugging through user-defined reductions: application to UML models. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*, Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer (Eds.). ACM, 87–97. https://doi.org/10.1145/3550355.3552447

[21] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. 2023. Temporal Breakpoints for Multiverse Debugging. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023, Cascais, Portugal, October 23-24, 2023*, João Saraiva, Thomas Degueule, and Elizabeth Scott (Eds.). ACM, 125–137. https://doi.org/10.1145/3623476.3623526

[22] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming* 60-61 (2004), 17 – 139.

[23] Guillaume Pothier, Éric Tanter, and José M. Piquer. 2007. Scalable omniscient debugging. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 535–552. https://doi.org/10.1145/1297027.1297067

[24] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art Science and Engineering of Programming* (07 2018). https://doi.org/10.22152/programming-journal.org/2019/3/1

[25] John R Searle. 1995. *The construction of social reality.* Simon and Schuster.

[26] J. Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (01 1991), 153–163. https://doi.org/10.1093/comjnl/34.2.153

[27] L. Thomas van Binsbergen. 2019. *Executable formal specification of programming languages with reusable components.* Ph. D. Dissertation. Royal Holloway, University of London, Egham, UK.

[28] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. 2020. eFLINT: A Domain-Specific Language for Executable Norm Specifications. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2020).* ACM.

[29] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoît Combemale, and Olivier Barais. 2020. A principled approach to REPL interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and*

Reflections on Programming and Software, Onward! 2020, Virtual, November, 2020. ACM, 84–100. https://doi.org/10.1145/3426428.3426917

[30] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. 2019. Executable Component-Based Semantics. *Journal of Logical and Algebraic Methods in Programming* 103 (feb 2019), 184–212. https://doi.org/10.1016/j.jlamp.2018.12.004

[31] Robert van Doesburg and Tom M. van Engers. 2016. Perspectives on the Formal Representation of the Interpretation of Norms. In *Legal Knowledge and Information Systems - JURIX 2016: The Twenty-Ninth Annual Conference (Frontiers in Artificial Intelligence and Applications, Vol. 294)*, Floris Bex and Serena Villata (Eds.). IOS Press, 183–186. https://doi.org/10.3233/978-1-61499-726-9-183

[32] Robert Michael Walsh. 2016. *Adapting Compiler Front Ends for Generalised Parsing*. Ph. D. Dissertation. Royal Holloway, University of London.

Received 4 March 2025; accepted 9 April 2025