CUTECat: Concolic Execution
for Computational Law

Pierre Goutagny[1]([envelope]) [iD], Aymeric Fromherz[2] [iD], and Raphaël Monat[1] [iD]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, 59000 Lille, France
pierre.goutagny@inria.fr, raphael.monat@inria.fr
[2] Inria Paris, Paris, France
aymeric.fromherz@inria.fr

Abstract. Many legal computations, including the amount of tax owed by a citizen, whether they are eligible to social benefits, or the wages due to civil state servants, are specified by *computational laws*. Their application, however, is performed by expert computer programs intended to faithfully transcribe the law into computer code. Bugs in these programs can lead to dramatic societal impact, e.g., paying employees incorrect amounts, or not awarding benefits to families in need.

To address this issue, we consider concolic unit testing, a combination of concrete execution with SMT-based symbolic execution, and propose CUTECat, a concolic execution tool targeting implementations of computational laws. Such laws typically follow a pattern where a base case is later refined by many exceptions in following law articles, a pattern that can be formally modeled using *default logic*. We show how to handle default logic inside a concolic execution tool, and implement our approach in the context of Catala, a recent domain-specific language tailored to implement computational laws. We evaluate CUTECat on several programs, including the Catala implementation of the French housing benefits and Section 132 of the US tax code. We show that CUTECat can successfully generate hundreds of thousands of testcases covering all branches of these bodies of law. Through several heuristics, we improve CUTECat's scalability and usability, making the testcases understandable by lawyers and programmers alike. We believe CUTECat paves the way for the use of formal methods during legislative processes.

1   Introduction

Since at least the Sumerian empire and the code of Ur-Nammu [57], human societies have been governed by laws. From constitutional law to environmental law, including criminal law, immigration law or intellectual property law, laws are applied in a wide range of contexts, representative of the diverse facets of modern societies. Laws are typically stated in natural language, e.g., English, and therefore require human interpretation to determine when and how they must apply: to do so, a criminal trial might rely on a grand jury to determine the guilt of a defendant, while companies typically hire highly specialized lawyers to ensure that a merger is performed according to corporate law.

Other laws, however, focus on defining well-specified computations, for instance, the amount of taxes that a household owes, the wages of civil state servants, or whether a given family is eligible to social benefits depending on their situation. As they must be applied to a large number of citizens, such laws, commonly known as *computational laws*, are typically implemented in expert computer systems, which are able to automate the computation for any input.

Unfortunately, as for any computer program, legal expert systems are not immune to bugs, which can have tremendous consequences, either for the states or their citizens. Case in point, more than half of all Canadian civil servants suffered pay issues from the Phoenix automated payroll system, resulting in years of fixing incurred financial issues for civil servants, and, for the Canadian government "$2.2 billion in unplanned expenditures", while the program should have provided "$70 million in annual savings by centralizing pay operations" [40]. Similarly, bugs in Louvois, the French army payroll system, led to several years of incorrect payments to military households, either causing missed wages or overpayments that had to be reimbursed by individuals years later [42]. Such cases are not isolated incidents, and examples of public legal systems that were either faulty or abandoned during development abound in many countries [47].

One of the core issues lies with the particular structure of computational law, commonly consisting of a base case refined by exceptions spread out throughout law texts. This structure, corresponding to *default logic* [6, 28], is not straightforward to encode using modern programming languages. This leads to discrepancies between code and law, where design choices made by programmers can be hard to understand by lawyers. Conversely, maintaining such implementations when legislative processes modify the law becomes tricky, as programmers cannot necessarily accurately pinpoint required changes in their code.

To address this issue, Merigoux et al. [37] recently introduced Catala, a domain-specific language (DSL) tailored to implement computational laws. At the heart of Catala lies programming constructs called *default terms*, which faithfully implement default logic. Default terms are first-class citizens in the core Catala language. To simplify their use, they are exposed through a custom syntax designed to be understandable by programmers and lawyers alike. This syntax enables Catala programmers to define the base case of a default term and its corresponding exceptions in different parts of the code, thus accurately reflecting the structure of legal texts. By combining this feature with literate programming, Catala enables a programming style where an official legal text is intertwined with its implementation, thus allowing programmers to work in concert with lawyers to ensure a faithful translation into code of the law.

To minimize common programming issues, Catala's design purposely avoids risky programming languages constructs, e.g., by providing infinite-precision numbers rather than floating-point ones, and by avoiding NULL values. Despite this effort, legal implementations in Catala can nevertheless contain bugs for three reasons. First, by heavily operating on numerical values such as amounts of money, Catala is not immune to standard runtime errors such as divisions by zero, possibly leading to crashes. Second, in spite of its peer-programming

approach where lawyers and programmers work together to implement the law, translations of legal texts into code amount to legal interpretations, which can be erroneous. Last, the law itself might exhibit inconsistencies, for instance, by defining contradicting exceptions under specific circumstances, or by forgetting to consider some situations – in Catala implementations, this would lead to a runtime exception being raised by the program.

To circumvent these issues, public administrations heavily rely on testing; some departments are in charge of manually handcrafting testcases, interpreting the law to compute results, and comparing them to the outputs of legal expert systems to ensure their conformance with legal texts. Unfortunately, the high number of cases, and the regular modifications of laws make this manual process costly and painstaking, and do not guarantee that corner cases are not missed.

To address this problem, this paper therefore advocates applying formal verification techniques to computational law. Doing so raises several challenges. First, formal verification tools must be able to efficiently reason about the core concept of computational law, namely, default logic. Second, to facilitate its adoption, the verification process must not require expert knowledge of formal verification, and thus needs to be as automated as possible. Last but not least, when identifying issues, it is crucial that verification tools provide concrete examples usable by lawyers, so that they can compare them with legal texts and determine whether these correspond to law inconsistencies.

Given these constraints, we focus our attention on the application of concolic unit testing [21, 51], a combination of concrete execution with SMT-based symbolic execution, which we believe to hit a sweet spot when it comes to reasoning about computational law. Computational law, and thus its implementation in Catala, is deterministic, and does not contain loops or recursion, alleviating a well-known challenge for concolic testing [10, 20]. Additionally, its automated nature and the concrete inputs it generates during analysis pave the way for use by lawyers and programmers alike.

Our contributions are the following: we show how to concolically execute computational law programs by providing a formal concolic semantics for default terms (Sec. 2.2). Relying on this formal model, we then implement CUTECat, a concolic execution execution engine for Catala programs (Sec. 3). CUTECat aims to detect all runtime errors, such as divisions by zero but also law inconsistencies due to missing, or conflicting interpretations. CUTECat includes several optimizations, both to improve its performance and scalability, as well as its usability, aiming to generate human-friendly testcases to facilitate legal interpretation by lawyers (Sec. 4). Finally, we experimentally evaluate CUTECat on a range of Catala programs (Sec. 5). Relying on real-world codebases from French and US laws, we first perform an ablation study to evaluate the impact of our different optimizations. We then conclude by empirically demonstrating that CUTECat can scale to the largest real-world Catala codebase currently available, namely, the implementation of the French housing benefits (19655 lines of Catala, including specification), generating 186390 tests in less than 7 hours of

CPU time. Our development is open-source and publicly available on GitHub; all experimental claims made in this paper are documented in an artifact [23].

2   Encoding Default Logic in Concolic Execution

In this section, we present our encoding of default logic into concolic execution. We start with some background about computational law, and how it relates to default logic.

2.1   Computational Law and Default Logic

Throughout this paper, we will rely on a simplified income tax computation as a running example. This computation determines the amount of taxes a household must pay, depending on their income. Typically, regulations specify a tax rate, that is multiplied by the household income to determine the income tax to pay. For instance, if the tax rate is set to 20%, a household earning $100,000 would pay $20,000 in income tax. In practice, tax laws commonly define different tax rates for different income brackets; we omit this in our example for simplicity.

To address several societal issues, this computation can however be modified depending on households. For instance, low-income households (which we define in this example as earning no more than $10,000) might be taxed to a reduced rate of 10%. Furthermore, large households (which we define in this example as households with three or more children), might benefit from a tax break, for instance, lowering their tax rate to 15%. For now, we do not set any interpretation priority for households having both a low income and a large number of children.

This structure is typical of computational laws: we have one *base case* (the tax rate is 20%), followed by several *conditional exceptions*, capturing specific situations (large, or low-income households). Implementing this structure in traditional programming languages is challenging for two reasons. First, exceptions in legal texts are commonly spread out through several law articles, which does not match the standard monolithic variable or function definitions. Second, legal implementations should closely follow the structure of the law they implement. Implementations correspond to applications of the law; any difference with the original text thus amounts to a legal reinterpretation, which should be performed by lawyers, and not programmers. Heavily transforming the law to match traditional programming idioms is therefore ill-advised. To address this issue, earlier work proposed *default terms* [37], heavily inspired by default logic [6, 28].

Default terms take the form $\langle e_1, \ldots, e_n \mid e_{just} :- e_{cons}\rangle$, where $e_1, \ldots, e_n$ are default expressions called 'exceptions', e_{just} is a boolean expression called 'default condition', and e_{cons} is a default expression. Empty values are written as $\varnothing$, and conflict values, when two exceptions occur at the same time, as $\circledast$. Due to typing guarantees, omitted here for brevity, the default condition never reduces to $\varnothing$ or $\circledast$. The grammar and formal semantics of default terms, adapted from [37], are available in Fig. 1 and Fig. 2 respectively. Expressions and values also contain numeric and boolean expressions, whose semantics is standard and

$$\text{Value} \quad v ::= \texttt{true} \mid \texttt{false} \mid n \qquad \text{boolean, integer literals}$$
$$\mid \varnothing \qquad\qquad\qquad\qquad \text{empty value}$$
$$\mid \circledast \qquad\qquad\qquad\qquad \text{conflict value}$$

$$\text{Expression } e ::= x \mid v \qquad\qquad\qquad\qquad \text{variable, values}$$
$$\mid e \bowtie e \qquad \text{integer, boolean binary operators}$$
$$\mid \langle \overrightarrow{e} \mid e :\text{-} e \rangle \qquad\qquad\qquad \text{default term}$$

Fig. 1: Grammar of default terms

DefaultExpr
$$\frac{e \longrightarrow^* v \qquad v \neq \circledast}{\langle v_1, \ldots, v_i, e, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow \langle v_1, \ldots, v_i, v, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle}$$

DefaultError
$$\frac{e \longrightarrow^* \circledast}{\langle v_1, \ldots, v_i, e, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow \circledast}$$

DefaultTrueNoExceptions
$$\frac{e_{just} \longrightarrow^* \texttt{true}}{\langle \varnothing, \ldots, \varnothing \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow e_{cons}}$$

DefaultFalseNoExceptions
$$\frac{e_{just} \longrightarrow^* \texttt{false}}{\langle \varnothing, \ldots, \varnothing \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow \varnothing}$$

DefaultOneException
$$\frac{v \neq \varnothing, \circledast}{\langle \varnothing, \ldots, \varnothing, v, \varnothing, \ldots, \varnothing \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow v}$$

DefaultExceptionsConflict
$$\frac{v_i \neq \varnothing \qquad v_j \neq \varnothing \qquad \forall k, v_k \neq \circledast}{\langle v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow \circledast}$$

Fig. 2: Selected reduction rules for default terms

thus omitted from the presentation. According to this semantics, default terms are executed as follows:

- If all exceptions reduce to empty values, that is, if no exception is raised or if there are none, then the default expression reduces either to e_{cons} if condition e_{just} evaluates to true (DefaultTrueNoExceptions), or to $\varnothing$ otherwise (DefaultFalseNoExceptions). For instance, the term $\langle \mid \texttt{false} :\text{-} 1 \rangle$ reduces to $\varnothing$.
- If exactly one exception e_i reduces to a non-empty value, then the default expression reduces to its value (DefaultOneException). Therefore, the term $\langle \langle \mid \texttt{true} :\text{-} 1 \rangle, \langle \mid \texttt{false} :\text{-} 2 \rangle \mid \texttt{true} :\text{-} 3 \rangle$ reduces to 1.
- If more than one exception expression reduces to a non-empty value, that is if several exceptions are raised at the same time, then the default expression reduces to a conflict error $\circledast$ (DefaultExceptionsConflict)[3]. Thus, $\langle \langle \mid \texttt{true} :\text{-} 1 \rangle, \langle \mid \texttt{true} :\text{-} 2 \rangle \mid \texttt{true} :\text{-} 3 \rangle$ yields $\circledast$.

Remark 1. To lighten notations, we will omit the list of exceptions from a default term when it is empty. For instance, the term $\langle \mid e_1 :\text{-} e_2 \rangle$ will be written as $\langle e_1 :\text{-} e_2 \rangle$.

---

[3] Two raised exceptions are considered a conflict even if they lead to the same result.

Encoding the income tax rate we described earlier as a default term is then straightforward. As the base case always applies, the default condition e_{just} is trivial, and the two cases for low-income and large households are encoded as exceptions, leading to the following term:

$$\langle\langle \texttt{income} \leq \$10,000 \ \texttt{:- } 10\% \rangle, \langle \texttt{nb\_children} \geq 3 \ \texttt{:- } 15\% \rangle \ | \ \texttt{true} \ \texttt{:- } 20\% \rangle$$

2.2   Concolically Executing Default Terms

We now present how to support default terms during concolic execution. We start with some preliminaries about concolic execution, and the related concept of symbolic execution.

Background: Symbolic and Concolic Execution. Symbolic execution [5, 14, 24, 27] is a program analysis technique that aims to explore all feasible program paths in a program, and that has been successfully applied to a wide range of languages [8, 9, 13, 16, 39, 45, 46]. In symbolic execution, concrete inputs are replaced by symbolic values, and a symbolic interpreter is tasked with executing the program under test. As the program is executed, the interpreter will collect symbolic constraints characterizing a given program execution path through a symbolic *path constraint*. When hitting a conditional branching, the symbolic interpreter will add the condition to the current path constraint, query an SMT solver [2, 4, 15, 18] to determine whether the path constraint is satisfiable, and resume execution of the program. When a constraint is deemed unsatisfiable, the interpreter will backtrack, and attempt to explore another execution path.

To make things more concrete, consider the small program $P \overset{\text{def}}{=}$ `if x > 0 then return 0 else if y < 10 then return 1 else error`. We present the different steps of the symbolic execution in Fig. 3, showcasing the generated path constraint tree at different steps. Treating x and y as symbolic variables, a symbolic interpreter would first reach the conditional branching `if x > 0`, and generate two paths to explore, corresponding to the path conditions $x > 0$ and $\neg(x > 0)$ (Fig. 3a). As both path conditions are satisfiable, both execution paths need to be explored. Arbitrarily picking the first path condition, it would then reach the terminal statement `return 0`, terminating the execution of this path (Fig. 3b). Executing the second path, it would then split the path constraint tree again according to the constraint $y < 10$. As both path conditions $\neg(x > 0), y < 10$ and $\neg(x > 0), \neg(y < 10)$ are satisfiable, the symbolic interpreter would explore both paths, thus detecting that the error statement is reachable (Fig. 3c).

Despite its successes, symbolic execution however faces several limitations. Symbolic execution tools heavily rely on SMT solvers, and therefore struggle when considering symbolic constraints beyond theories well-supported by automated solvers, such as non-linear arithmetic. Additionally, symbolic interpreters typically require access to the analyzed code, and hence do not work well when interacting with an external environment or external library calls [1, 10, 20].

To address these issues, *concolic execution*, a combination of *conc*rete and symb*olic* execution was proposed [21, 51]. In concolic execution, the program is

(a) First step   (b) Second step
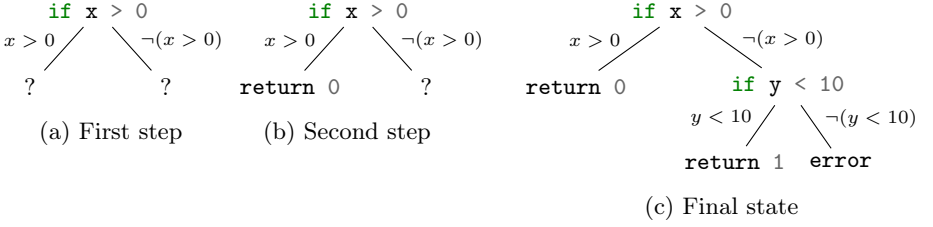
(c) Final state

Fig. 3: Path constraint trees at different stages of symbolic execution. Nodes marked as ? are not yet explored

executed with concrete inputs, and instrumented to collect symbolic constraints during execution. When one execution terminates, some of the constraints in the path condition are negated and new inputs are generated through a call to an SMT solver, leading to the exploration of a new program path.

Compared to symbolic execution, concolic execution has several advantages. First, when faced with unsupported logical theories, concolic execution can still make progress by executing the program, instead of getting stuck when encountering complex constraints. Second, by relying on concrete inputs covering a wide range of program paths, concolic execution naturally provides a reproducible testbed that can be used for automated unit testing, even when symbolic execution would be imprecise due to complex program statements or calls to libraries whose code is unavailable [20]. Last, while the generation of new inputs is typically performed by an automated solver to guarantee the exploration of new program paths, it can also be combined with other input generation techniques such as fuzzing [22, 32, 44, 54].

To formally describe concolic execution, we will rely on the following notations for instrumented semantics. When $e \longrightarrow^* e'$ denotes the concrete evaluation from expression e to expression e', $\chi \vdash e \longrightarrow^* \chi' \vdash e'$ will denote that, under initial path condition χ, e evaluates to e' with new path condition χ'.

Coming back to our toy program P, concolic testing would therefore proceed as follows. Let us assume that we start with inputs $x = 5, y = 5$. Then, we would have $\cdot \vdash P \longrightarrow^* x > 0 \vdash 0$, i.e., the concrete execution of the program returned the value 0, and the path condition $x > 0$ was collected. A concolic engine would then negate part of the path condition, query the SMT solver with the new constraints, i.e., $\neg(x > 0)$, and start another iteration with the inputs generated by the solver. This process would repeat until negating constraints does not lead to novel execution paths and all feasible program paths have been explored, with concrete inputs leading to each of these paths.

Encoding Default Terms. To concolically execute default terms, we now need to provide a symbolic representation to complement the concrete semantics presented in Fig. 2. To do so, we show how to instrument the concrete semantics to collect symbolic constraints characterizing the current execution path, by materializing the control flow structure of the evaluation of a default expression. In order to cover all program paths induced by default terms, this encoding must

therefore consider all combinations of raised exceptions, as well as a branching due to the default condition e_{just} when no exception is triggered.

We formally define our instrumented semantics in Fig. 4. We slightly extend the notations presented in the previous section, and denote as $\mathcal{X} \vdash e \mid C$ a default expression e with symbolic constraints $\mathcal{X}$ and C. The constraint $\mathcal{X}$ corresponds to the path condition collected up to the evaluation of the current default term. The constraint C, which we will call *local constraint*, corresponds to the constraints collected during the evaluation of the current term. The distinction between $\mathcal{X}$ and C is irrelevant for the rules presented in this section, and $\mathcal{X} \vdash e \mid C$ can safely be understood as $\mathcal{X} \wedge C \vdash e$, i.e., the current path condition is the conjunction of $\mathcal{X}$ and C. The importance of this separation will appear in Sec. 4.1, when discussing several optimizations to concolic execution of default terms. When either $\mathcal{X}$ or C are trivial, we will omit them to simplify notations.

To illustrate how our semantics operates, we will rely on the following example, where x and b are respectively integer and boolean variables: $\langle\langle b :\text{-} 1\rangle, \langle x = 0 :\text{-} 2\rangle \mid x > 0 :\text{-} 3\rangle$. Let us assume that we want to concolically execute this term with $x = 3$ and $b = \textbf{true}$. Initially, both the global constraints $\mathcal{X}$ and local constraints C are trivial. Following rule C-DEFAULTEXPR, we must first reduce the exception $\langle b :\text{-} 1\rangle$ to a value. Since we have a default term (cf. Remark 1) and its default condition b holds, we can apply rule C-DEFAULTTRUENOEXCEPTIONS, obtaining the concolic term $b \vdash 1$. We conclude the application of rule C-DEFAULTEXPR, leading to the concolic term $\langle 1, \langle x = 0 :\text{-} 2\rangle \mid x > 0 :\text{-} 3\rangle \mid b$. We then perform a similar reduction on the second exception by combining rules C-DEFAULTEXPR and C-DEFAULTFALSENOEXCEPTIONS to obtain the concolic term $\langle 1, \varnothing \mid x > 0 :\text{-} 3\rangle \mid b \wedge \neg(x = 0)$. As exactly one exception reduced to a non-empty or error value, the final step in the concolic execution is then to apply rule C-DEFAULTONEEXCEPTION; local constraints are thus appended to the current (empty) path condition. The execution therefore terminates with the value $b \wedge \neg(x = 0) \vdash 1$, accurately capturing that any set of inputs satisfying the path constraint $b \wedge \neg(x = 0)$ will follow the same execution path.

A Complete Concolic Execution. Equipped with our concolic semantics, we can now define a concolic execution operating on default terms. To do so, we follow the standard workflow of concolic execution: starting from an initial, arbitrary set of inputs, we concolically execute the program, and collect the corresponding path constraints. Once the execution terminates, we choose another unexplored path, querying the SMT solver to generate an input satisfying the new path constraint, and therefore leading to a different execution. We repeat this process until all paths have been explored, or an exploration timeout has been reached.

To illustrate how a complete concolic execution operates, we will reuse the default term $\langle\langle b :\text{-} 1\rangle, \langle x = 0 :\text{-} 2\rangle \mid x > 0 :\text{-} 3\rangle$ previously presented. Starting from inputs $x = 3$ and $b = \textbf{true}$, we previously saw that this term reduced to the value 1, generating the constraints b and $\neg(x = 0)$. By negating the last constraint and querying the solver, we obtain new inputs $x = 0$ and $b = \textbf{true}$, and perform another concolic iteration. Repeating this process leads to the generation of 5 different testcases fully covering the program, and identifying two

Notations

$$\boxed{\chi \vdash e \mid C}$$ "Expression e has path condition χ and local constraint C"

$$\boxed{\chi \vdash e}$$ "Syntactic sugar for $\chi \vdash e \mid \cdot$"

$$\boxed{e \mid C}$$ "Syntactic sugar for $\cdot \vdash e \mid C$"

$$\boxed{e}$$ "Syntactic sugar for $\cdot \vdash e \mid \cdot$"

$$\boxed{\chi \vdash e \mid C \longrightarrow \chi' \vdash e' \mid C'}$$ "Under path condition χ and local constraint C,
e evaluates to e' with new path condition χ' and local constraint C'"

C-DefaultExpr

$$\frac{e \longrightarrow^* \chi' \vdash v \qquad v \neq \circledast}{\chi \vdash \langle v_1, \ldots, v_i, e, \ldots \mid e_{just} \ \text{:-} \ e_{cons} \rangle \mid C \longrightarrow \chi \vdash \langle v_1, \ldots, v_i, v, \ldots \mid e_{just} \ \text{:-} \ e_{cons} \rangle \mid C \wedge \chi'}$$

     C-DefaultError

$$\frac{e \longrightarrow^* \chi' \vdash \circledast}{\chi \vdash \langle v_1, \ldots, v_i, e, \ldots, \mid e_{just} \ \text{:-} \ e_{cons} \rangle \mid C \longrightarrow \chi \wedge \chi' \wedge C \vdash \circledast}$$

     C-DefaultTrueNoExceptions

$$\frac{e_{just} \longrightarrow^* \chi' \vdash \text{true}}{\chi \vdash \langle \varnothing, \ldots, \varnothing \mid e_{just} \ \text{:-} \ e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \wedge \chi' \wedge e_{just} \vdash e_{cons}}$$

     C-DefaultFalseNoExceptions

$$\frac{e_{just} \longrightarrow^* \chi' \vdash \text{false}}{\chi \vdash \langle \varnothing, \ldots, \varnothing \mid e_{just} \ \text{:-} \ e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \wedge \chi' \wedge \neg(e_{just}) \vdash \varnothing}$$

     C-DefaultOneException

$$\frac{v \neq \varnothing, \circledast}{\chi \vdash \langle \varnothing, \ldots, \varnothing, v, \varnothing, \ldots, \varnothing \mid e_{just} \ \text{:-} \ e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \vdash v}$$

     C-DefaultExceptionsConflict

$$\frac{v_i \neq \varnothing \qquad v_j \neq \varnothing \qquad \forall k, v_k \neq \circledast}{\chi \vdash \langle v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n \mid e_{just} \ \text{:-} \ e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \vdash \circledast}$$

Fig. 4: Concolic semantics for default terms

execution paths leading to $\circledast$ and $\varnothing$ respectively. The summary of the concolic execution is available in Fig. 5.

3   CUTECat: Implementing Concolic Execution

Relying on the default logic and its concolic interpretation presented in the previous section, we now present CUTECat, a concolic execution engine for the Catala programming language.

     Catala is a recent domain-specific language tailored to implement computational laws, relying on default logic under the hood [37]. We show an example Catala program in Fig. 6, encompassing the default term for the simplified in-

Step	b	x	Output	Constraints after evaluation	Next constraints to try
# 1	**true**	**3**	**1**	$[(b), \neg(x=0)]$	$b \wedge x = 0$
# 2	**true**	**0**	⊛	$[(b), (x=0)]$	$\neg b$
# 3	**false**	**3**	**3**	$[\neg(b), \neg(x=0), (x>0)]$	$\neg b \wedge \neg(x=0) \wedge \neg(x>0)$
# 4	**false**	**−1**	∅	$[\neg(b), \neg(x=0), \neg(x>0)]$	$\neg b \wedge x = 0$
# 5	**false**	**0**	**2**	$[\neg(b), (x=0)]$	–

Fig. 5: Concolic testcase generation for $\langle\langle b \,:\!\!- 1\rangle, \langle x = 0 \,:\!\!- 2\rangle \mid x > 0 \,:\!\!- 3\rangle$

```catala
declaration structure Household:
 data income content money
 data nb_children content integer

declaration scope IncomeTaxComputation:
 input house content Household
 internal tax_rate content decimal
 output income_tax content money
```

Article 1

The income tax for an individual is
defined as a fixed percentage of the
individual's income over a year.
```catala
scope IncomeTaxComputation:
 definition income_tax equals
   house.income * tax_rate
```

Article 2

The fixed percentage mentioned at
article 1 is equal to 20%.
```catala
scope IncomeTaxComputation:
 definition tax_rate equals 20%
```

Article 3

If the individual's income is less
than $10,000, the fixed percentage
mentioned at article 1 is equal to 10%.
```catala
scope IncomeTaxComputation:
 exception definition tax_rate
  under condition house.income <= $10,000
  consequence equals 10%
```

Article 4

If the individual is in charge of 3 or
more children, then the fixed percentage
mentioned at article 1 is equal to 15%.
```catala
scope IncomeTaxComputation:
 exception definition tax_rate
  under condition house.nb_children >= 3
  consequence equals 15%
```

Fig. 6: Running example: a simplified income tax computation

come tax computation described in Sec. 2.1. One important aspect of Catala is
its literate programming capabilities, allowing to reflect the structure of the law
in the implementation. As seen in this example, each unit of law is immediately
followed by its implementation. The main purpose is to facilitate the maintain-
ability and transparency of the implementation with respect to the law test; at
compilation time, Catala will rely on the Markdown code markers ```catala to
extract the implementation. We refer the interested reader to the Catala tuto-
rial [36] for a more detailed description of the merits of literate programming
when implementing computational law.

We now provide an overview of the Catala language on our running example.
Catala requires strongly typed declarations exposing the interfaces of a program.
Here, lines 2-4 define a `Household` record, defined by an `income` and a number of
children. Scopes are the basic abstraction unit in Catala; they can be conceived
as a loose equivalent to functions in other programming languages. A scope
is first statically declared, with a typed declaration of its inputs, outputs and
internal variables. In our example, lines 6-9 define the interface of our income tax
computation, which computes an income tax for a given household. The main
case of the income tax computation is defined lines 17-19 and 26-27: the default
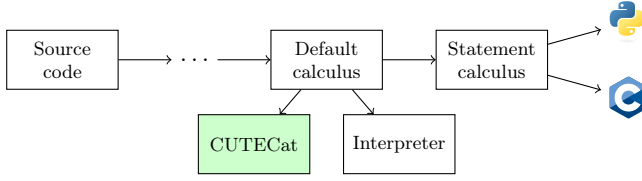
Fig. 7: Passes of the Catala compiler, and CUTECat integration

tax rate is 20% of the household's income. This rate is amended by two cases, one setting it at 10% for low incomes (lines 35-38), the second setting the rate to 15% for large families (lines 46-49). Note that while these new definitions are following one another in this simplified example, real-world Catala implementations might have amended these definitions in several different files, for instance if the base case is defined in the Tax Code while the large families exception is defined in the Family Code. During compilation, the Catala toolchain will combine the three definitions of `tax_rate` in `IncomeTaxComputation` through the use of a default term, corresponding to the one defined in the previous section: $\langle\langle$`income` $\leq$ $\$10,000$:- $10\%\rangle, \langle$`nb_children` ≥ 3 :- $15\%\rangle$ | `true` :- $20\%\rangle$

Our concolic engine, CUTECat, has been integrated directly into the Catala toolchain, whose relevant parts are shown in Fig. 7. To allow integration of Catala implementations in existing projects, the compiler translates Catala source code into mainstream programming languages like C or Python. To do so, it relies on a series of intermediate representations (IRs), elided here, until reaching the default calculus. The default calculus is one of the last IRs in the compilation pipeline, and is very close to the statement calculus which transforms expressions into statements before emitting C or Python code.

Default terms are explicitly materialized in the default calculus; it is where the reference Catala interpreter operates, following the semantics for default terms described in Fig. 2. This is therefore a natural choice for the implementation of our concolic engine; additionally, being located at the same compilation step of the interpreter nullifies any discrepancies of potential bugs introduced earlier in the compilation pipeline. We implemented our concolic interpreter as a fork of the standard interpreter, instrumenting it to collect symbolic constraints. This design allows us to closely follow the reference semantics of Catala, thus minimizing implementation mistakes, while also simplifying the maintenance of CUTECat when the Catala language evolves.

Similarly to other works on concolic execution [32, 51], CUTECat relies on a depth-first search (DFS) strategy to explore new execution paths. In a DFS exploration, a concolic engine generates new inputs by negating the last constraint added to the path condition. Concretely, during a concolic iteration, CUTECat keeps the constraints C used to generate the current input. Each constraint is annotated to indicate whether it has already been negated. The execution generates a path condition C', which is compared to C; all constraints in $C' \setminus C$ are marked as new, and the last new constraint in C' is negated to generate a new input. This workflow corresponds to a DFS exploration of the path constraint tree;

```
1    scope IncomeTaxComputation:
2     exception definition income_tax
3      under condition house.income <= $10,000
4      consequence equals house.income * 10%
```

Fig. 8: Removing conflicts in income tax computation

however, it only requires keeping the last explored branch in memory instead of the entire tree, therefore improving the performance of concolic execution.

To illustrate how CUTECat operates, we now describe the concolic execution of our running example. CUTECat generates here four different cases, in an order determined by the values used in the first iteration; in our case, they are set to income = $0 and nb_children = 0.

1. income is $0 and there are no children. In that case, only the first exception (lines 35-38 of Fig. 6) can be applied, and the tax is $0.
2. income is $0 and there are three children. In that case, both exceptions (lines 35-38 and 46-49) can be applied. A conflict is raised to the user, showing that the law has been unfaithfully translated, or that it is ambiguous.
3. income is $10,000.01 and there are two children. In that case, the general case (lines 26-27) applies, and the tax is $2,000 (monetary amounts are rounded to the cent).
4. income is $10,000.01 and there are three children. In that case, the exceptional case at lines 46-49 applies and the tax is $1,500.

By default, CUTECat outputs these different cases directly to the user, however, we also provide facilities to directly generate Catala *test scopes*. These test scopes can be saved, and easily replayed using the Catala toolchain, either by using the reference interpreter or by compiling it to one of the Catala backends.

As identified by our concolic execution, our running example can raise conflicts, i.e., when considering large, low-income households. This conflict is not due to an implementation error, but rather to an ambiguity in our (simplified) law. Resolving this ambiguity requires legal interpretation, which must be performed by lawyers, and not programmers. In this case, legal precedents might suggest that ambiguities shall be resolved in the most favorable way to citizens, therefore defining a 10% tax rate. To implement this decision, programmers could thus modify the Catala implementation of Article 3 as shown in Fig. 8, by materializing a priority order on the evaluation of both exceptions: as the exception is now defined on income_tax, this exception will trigger first, before reaching the base computation depending on tax_rate[4].

CUTECat is implemented in 3,400 lines of OCaml code, and relies on the Z3 SMT solver [18], called through its OCaml bindings. It is an integral part of the Catala toolchain, and therefore leverages its build system, simplifying the use of CUTECat on Catala projects. To provide users with more control on the concolic execution, CUTECat allows the use of assertions in the source code to restrict

---

[4] Note that a more idiomatic fix would make use of Catala "labels", that enable programmers to order exceptions explicitly. We omit this solution to keep our presentation of Catala small. The Catala tutorial [36] has more information on labels.

the input space, for instance, to specify that only households in a given country or state must be considered. Users can provide hints about initial inputs for the concolic execution, in order to quickly explore variants of a given situation.

Divisions and Rounding. In addition to our handling of default terms described in Sec. 2, CUTECat supports the concolic execution of most Catala constructs, including arbitrary-precision integers, booleans, money and decimal expressions; structures and field operations; algebraic data types and pattern-matching; functions and function calls; and conditionals. The concolic execution of most of these operations is standard, and we therefore omit their presentation. Two points are however of particular interest, namely, arithmetic divisions and rounding.

Divisions by zero are a well-known source of runtime errors. To reason about them, concolic tools therefore consider divisions as implicitly branching, and collect whether the denominator is equal to zero as a symbolic constraint.[5]

By providing different representations for numerical values, i.e., decimal (represented as rationals) and integers values, Catala also requires conversions between these types; when casting a decimal to an integer, this is implemented as a rounding operator. Catala's rounding convention is to round to the nearest integer; when two are equidistant, it returns to the one furthest away from 0. To model this semantics, we define a custom *round* Z3 function, defined as `round(q) = if q>=0 then floor(q + 1/2) else -floor(-q + 1/2)`; the `floor` operation is implemented using Z3's native casting from rationals to integers.

As part of our experimental evaluation of CUTECat (Sec. 5), our precise modeling of rounding led us to find inputs where the different Catala backends were inconsistent: rounding operations in Catala's Python backend exhibited minor differences with the reference interpreter. We reported this issue to the Catala developers, and upstreamed a fix to the compiler.

Limitations. CUTECat currently supports a large subset of Catala which is sufficient to analyze real-world programs (Sec. 5) that mainly rely on integer-rational reasoning as well as algebraic datatypes. However, operations on lists and dates are only partially supported. List operations in Catala include filtering depending on a condition, aggregating over list contents to compute all sorts of values, and applying a function to all elements in the list; providing symbolic encoding of such operations would require higher-order symbolic reasoning [43, 55]. On the other hand, dates can easily be represented in an SMT solver as the number of days since a specific point in time, e.g., the Unix epoch. This representation allows to model several operations, such as the addition of a (possibly symbolic) number of days, or the duration corresponding to the difference between two

---

[5] Note that our semantics conflates runtime errors with conflicts, and returns ⊛ in both cases. Indeed, the conflict value acts as a *de facto* uncatchable exception: when it appears, it stops the evaluation of the program, which immediately returns. From the user's perspective, the main concern is whether the program can lead to an error, independently of which one. As we generate concrete inputs, it is always possible to run the reference Catala interpreter to provide precise error reporting.

dates. Unfortunately, other operations, including month and year addition or returning the first day of a week or a month, have a more complex semantics [41] which does not naturally fit into this SMT encoding.

To handle these operations, we leverage the strengths of concolic execution over symbolic execution, and do not generate any symbolic encoding, relying instead entirely on concrete evaluation at the cost of completeness. We leave the exploration of suitable symbolic models for these operations to future work.

Finally, our implementation currently only supports the Z3 SMT solver, as it directly provides ready-to-use OCaml bindings. This limitation could be easily lifted, as we can already generate SMT-LIB files [3] for each SMT query, which can be consumed by most other SMT solvers [2, 4, 15]. Additionally, our prototype is single-threaded; the recent introduction of Multicore OCaml [52] nevertheless paves the way for parallelizing concolic execution [7, 53].

4   Improving the Scalability and Usability of CUTECat

To improve the scalability of CUTECat, we now discuss various optimizations and heuristics to enhance the performance of concolic execution, as well as the usability of our toolchain by non-expert users. The concrete impact of these optimizations will be evaluated experimentally in Sec. 5.

4.1   Optimizing Concolic Execution of Default Terms

The approach presented in Sec. 2.2 allows us to perform concolic execution on default terms. However, to improve interactivity with developers, our goal is not only to exhaustively analyze all program execution paths, but also to find possible errors as early as possible during the analysis. In this section, we therefore propose several optimizations aiming to prune constraints leading to redundant cases, and to prioritize executions that might lead to conflict errors.

Lazily Evaluating Exceptions. When evaluating a default term, the semantics presented in Fig. 2 requires first evaluating all exceptions, except if one of them raises a conflict error (DEFAULTEXPR). If two exceptions evaluate to a non-empty value, the ensuing conflict is therefore only detected when applying rule DEFAULTEXCEPTIONSCONFLICT. As concrete executions are in practice very fast, this has little performance impact when concretely executing a default term; furthermore, from a usability perspective, it is helpful to report all conflicting cases to users. However, the overhead becomes larger during concolic execution, where we must explore all possible execution paths when evaluating remaining exceptions, including many calls to an SMT solver.

To circumvent this issue, we propose in Fig. 9 alternative semantic rules to replace DEFAULTEXPR and DEFAULTEXCEPTIONSCONFLICT, as well as their concolic counterparts. These new rules stop the evaluation as soon as two exceptions return a non-empty value. This prevents exploring the remaining exceptions; the multiple execution paths would all lead to a conflict in the current default term.

DEFAULTEXPR-LAZY
$$\frac{e \longrightarrow^* v \qquad v \neq \circledast}{\langle \varnothing, \ldots, \varnothing, e, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow \langle \varnothing, \ldots, \varnothing, v, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle}$$

DEFAULTEXPRONE-LAZY
$$\frac{e \longrightarrow^* \varnothing \qquad v_i \neq \circledast, \varnothing}{\langle \varnothing, \ldots, v_i, \ldots, \varnothing, e, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow \langle \varnothing, \ldots, v_i, \ldots, \varnothing, \varnothing, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle}$$

DEFAULTEXCEPTIONSCONFLICT-LAZY
$$\frac{e \longrightarrow^* v \qquad v \neq \circledast, \varnothing \qquad v_i \neq \circledast, \varnothing}{\langle \varnothing, \ldots, v_i, \ldots, \varnothing, e, \ldots, \mid e_{just} :\text{-} e_{cons} \rangle \longrightarrow \circledast}$$

C-DEFAULTEXPR-LAZY
$$\frac{e \longrightarrow^* \chi' \vdash v \qquad v \neq \circledast}{\chi \vdash \langle \varnothing, \ldots, \varnothing, e, \ldots \mid e_{just} :\text{-} e_{cons} \rangle \mid C \longrightarrow \chi \vdash \langle \varnothing, \ldots, \varnothing, v, \ldots \mid e_{just} :\text{-} e_{cons} \rangle \mid C \wedge \chi'}$$

C-DEFAULTEXPRONE-LAZY
$$\frac{e \longrightarrow^* \chi' \vdash \varnothing \qquad v_i \neq \circledast, \varnothing}{\chi \vdash \langle \varnothing, \ldots, v_i, \ldots, \varnothing, e, \ldots \mid e_{just} :\text{-} e_{cons} \rangle \mid C \longrightarrow \chi \vdash \langle \varnothing, \ldots, v_i, \ldots, \varnothing, \varnothing, \ldots \mid e_{just} :\text{-} e_{cons} \rangle \mid C \wedge \chi'}$$

C-DEFAULTEXCEPTIONSCONFLICT-LAZY
$$\frac{e \longrightarrow^* \chi' \vdash v \qquad v \neq \circledast, \varnothing \qquad v_i \neq \circledast, \varnothing}{\chi \vdash \langle \varnothing, \ldots, v_i, \ldots, \varnothing, e, \ldots \mid e_{just} :\text{-} e_{cons} \rangle \mid C \longrightarrow \chi \wedge \chi' \wedge C \vdash \circledast}$$

Fig. 9: Alternative concrete and concolic lazy semantics for default terms

Consider for instance a default term $e := \langle e_1, e_2, e_3 \mid e_{just} :\text{-} e_{cons} \rangle$, where e_1 and e_2 both evaluate to values that are neither empty nor a conflict, respectively generating constraints C_1 and C_2. By following rule C-DEFAULTEXPR, concolically executing e would require concolically executing e_3 and generating its associated constraint C_3, which would then be added to the path condition when applying rule C-DEFAULTEXCEPTIONSCONFLICT. Further iterations of concolic execution would therefore negate the last constraints in the path condition, i.e., those in C_3, while preserving C_1 and C_2. As any input satisfying $C_1 \wedge C_2$ leads to e_1 and e_2 evaluating to non-empty nor conflict values, thus raising a conflict, this approach would induce a number of redundant iterations that is exponential in the number of constraints in C_3. By instead adopting rule C-DEFAULTEXCEPTIONSCONFLICT-LAZY and stopping the execution after the evaluation of e_2, we therefore prune C_3 from the path condition, thus reducing the number of iterations needed.

Reorganizing Exceptions. By relying on the semantics presented in Fig. 9, concolic executions of default terms can therefore be greatly shortened if exceptions evaluating to non-empty values are at the front of the exception list. Beyond not evaluating additional exceptions, their corresponding symbolic constraints will not be added to the path condition, thus reducing the size of the constraint tree and the number of iterations needed for concolic execution to terminate.

To leverage this fact, one key observation is that the result of the evaluation of a default term is independent of the evaluation order of the exception list. We formalize this property in Th. 1, which states that swapping any two exceptions

C-DefaultError-Early
$$\dfrac{e \longrightarrow^* \chi' \vdash \circledast}{\chi \vdash \langle v_1, \ldots, v_i, e, \ldots, \mid e_{just} \; \text{:-} \; e_{cons}\rangle \mid C \longrightarrow \chi \wedge \chi' \vdash \circledast}$$

Fig. 10: Alternative concolic semantics for conflict-inducing evaluations

results in evaluating the default term to the same value. This therefore allows to use different evaluation orders depending on the program and current symbolic state. In our implementation, which we describe in more detail in the next sections, we permute exceptions to group them depending on their free variables, a technique reminiscent of constraint reordering introduced by Cadar et al. [8].

Theorem 1 (Independence of the exception evaluation order). *If there exists a default value v such that $\langle \ldots, e_i, \ldots, e_j, \ldots \mid e_{just} \; \text{:-} \; e_{cons}\rangle \longrightarrow^* v$, then $\langle \ldots, e_j, \ldots, e_i, \ldots \mid e_{just} \; \text{:-} \; e_{cons}\rangle \longrightarrow^* v$*

Pruning Non-Conflict Path Conditions. As a last optimization, we present in Fig. 10 an alternate formulation of rule C-DefaultError, which applies when evaluating an exception yields a conflict value. If $\chi \vdash \langle e_1, \ldots, e_i, e, \ldots \mid e_{just} \; \text{:-} \; e_{cons}\rangle \longrightarrow^* \chi \vdash \langle v_1, \ldots, v_i, e, \ldots \mid e_{just} \; \text{:-} \; e_{cons}\rangle \mid C$, and $e \longrightarrow^* \chi' \vdash \circledast$, then the application of rule C-DefaultError will add both C and χ' to the path condition. We observe however that this will lead to redundant executions, as any input satisfying χ' will lead to e evaluating to $\circledast$. Pruning C from the path condition thus avoids unneeded concolic exploration, which we formally capture through rule C-DefaultError-Early.

This rule directly derives from Th. 1; indeed, the rule C-DefaultError-Early is conceptually equivalent to reorganizing the exception list to place e at the head of the list, and then applying C-DefaultError. This is where the need for differentiating between the path condition χ and local constraint C arises: while constraints corresponding to the evaluation of earlier exceptions can be safely dropped, the path condition up to this default term must be preserved.

Implementation. CUTECat implements the lazy evaluation of exceptions, as well as exception reorganization to order exceptions according to their sets of free variables. Pruning non-conflict path conditions is trickier to implement while following a DFS exploration strategy. A key invariant of DFS exploration is that, during a concolic iteration, the previous path condition (used to generate the current input) is a prefix of the current path condition up to the negated constraint. This does not hold with rule C-DefaultError-Early, as the local constraint C is dropped from the path condition. We leave the study of alternative exploration strategies and their impact as future work.

4.2   Generating Human-compatible Testcases

While many testcases might be equivalent from a semantic perspective (i.e., following the same program path), some might be easier to review by lawyers

and humans in general. Consider for instance the constraint $\neg(x \leq \$10,000)$, used in our running example to determine whether a household is in the low-income category. As we have seen in Sec. 3, Z3 might generate $x = \$10,000.01$ as a testcase. When crafting testcases, lawyers would instead lean towards round numbers, e.g., $x = \$11,000$, which would be easier to use when redoing manual computations of legal statutes to compare them to the implementation's output. To mimick this behavior, CUTECat therefore provides heuristics attempting to generate semantically equivalent, "human-friendlier" testcases.

To do so, we rely on the encoding of optional constraints in the solver, dubbed "soft constraints" [59]. Z3 natively allows to combine SMT reasoning with solving optimization objectives, including the specification of soft constraints; however, we encountered significant slowdowns when attempting to use this feature, raising scalability issues, that have been reported to Z3 developers. We posit that this is due to non-linearity in our soft-constraints – e.g., when specifying $x\%100 = 0$ – and in our rounding function defined in Sec. 3. We instead adopt an alternative approach, by implementing custom soft constraints through multiple queries to the solver. Concretely, when an iteration of concolic execution terminates and a new path condition C is generated, we first query Z3 to check whether C is satisfiable. If so, we further query Z3 by iteratively refining C with additional constraints forcing human-friendlier input generation.

Our current soft constraints particularly target monetary inputs. We first attempt to force such inputs to be multiples of 100. If impossible, we then try with multiples of 10, and finally with integers, aiming to prevent the use of cents. Empirically, these constraints have a significant impact: as monetary amounts are encoded at the cent level, Z3 frequently returns instances with cents, which are less readable and less pleasant to use when computing by hand, especially when percentages and rounding are involved. This paves the way for further improvements to CUTECat's usability; beyond monetary inputs, we envision the development of a lawyer-friendly custom configuration language, enabling the specification of preferred soft constraints depending on needs and usecases.

4.3 Pattern-Matching Case Folding

The Catala language supports simple pattern matching on the constructors of algebraic datatypes. By default, CUTECat's interpretation of pattern matching considers each case of the pattern matching as a different branch. That way, the logic of the pattern matching is materialized in the constraint tree itself, and the engine is tasked with successively trying each branch. However, we have noticed that this approach can be highly inefficient in existing Catala codebases.

Consider for example the snippet in Fig. 11, extracted from the Catala implementation of the French housing benefits. At lines 5-13, this program performs a match on the 9 variants of a sum type representing geographic areas of France (mainland, or overseas territories). To analyze this program, a naive implementation of our concolic interpreter would therefore consider 9 different, disjoint branches, leading to 9 distinct iterations.

```
1    scope HousingBenefitsForRentals:          8        -- Mayotte: true
2      exception definition family_rate         9        -- SaintBarthélemy: true
3      under condition                         10        -- SaintMartin: true
4        (match area with                      11        -- Guyane: false
5          -- Guadeloupe: true                 12        -- Métropole: false
6          -- Martinique: true                 13        -- SaintPierreEtMiquelon: false)
7          -- LaRéunion: true                  14    consequence equals ...
```

Fig. 11: French housing benefits case where pattern case folding is beneficial

Here however, the arms of several cases are identical; patterns can be conjoined to only create 2 branches, one corresponding to cases returning `true`, the other to cases returning `false`. Doing so heavily reduces the size of the path constraint tree; in more realistic examples, performance gains can be significant when such patterns appear deep inside expressions, requiring long concolic executions. We implemented a case folding optimization for pattern matching inside CUTECat, folding similar cases by considering the disjunction of their patterns.

4.4   SMT-Solving Optimizations

Trivial constraint simplification. The path constraints taken into account during the concolic execution can sometimes be trivial. Indeed, some conditions do not depend on input variables, or are compilation artifacts from the translation of the source code into the default calculus, where tautological conditions are sometimes added in default expressions. Negating these trivially true constraints in the hope of exploring other execution paths is pointless, as the new constraint given to the solver will be trivially false, and therefore unsatisfiable.

To avoid this, we query the SMT solver to simplify constraints before they are added in the path, and then remove trivial constraints statically. This avoids spurious calls to the SMT solver. Additionally, we perform a linear check for trivial unsatisfiability, by checking whether the newest constraint is a negation of a previous one. This last technique is well known by the concolic solver community, and already mentioned in the seminal work of Sen et al. [51].

Incremental SMT solving. Z3 provides incremental solving capabilities, where sets of constraints can be incrementally added or removed through stack-based `push` and `pop` operators. When popping the stack, the solver backtracks to an earlier proof state, only removing lemmas learned between `push` and `pop` and avoiding reproving many facts. We implemented support for incremental solving, pushing a new stack for each element in the path condition. As CUTECat's exploration strategy is currently fixed to depth-first search (DFS), incremental solving is particularly appealing: constraints are naturally changed in a last-in-first-out fashion, corresponding to a stack-based data structure.

5   Evaluating CUTECat

We now turn to our experimental evaluation of CUTECat. We first quantify the impact of the optimizations described in Sec. 4, relying on four Catala projects

predating this work (Sec. 5.1). We then discuss CUTECat's ability to identify conflicts (Sec. 5.2), before evaluating CUTECat on the largest Catala program up to this day, namely, an implementation of the French Housing Benefits (Sec. 5.3). All experiments have been performed on a desktop machine featuring an Intel Core i7-12700 and 128GB of DDR5 RAM (although we have never seen CUTE-Cat use more than 1GB), running Ubuntu 24.04.1, OCaml 4.14.1 and Z3 4.12.5.

5.1 Evaluating CUTECat's Optimizations

Benchmarks Overview. To evaluate the performance of CUTECat and its optimizations, we have selected four different, real-world Catala programs as benchmarks for our evaluation. These programs predate our implementation of CUTE-Cat, and have been written by third-party, experienced Catala developers. One benchmark corresponds to the implementation of a fragment of Section 132 of the US Tax Code, defining qualified employee discounts. Other benchmarks implement diverse bodies of law in France: one codifies minimum salary (SMIC), another the computation of a family quotient used in the income tax computation (Family Quotient). The last benchmark computes the housing benefits in the rental case, where additional restrictions have been made: households are located in mainland France and contain less than 10 children. These benchmarks reflect multiple facets of the law, and therefore exhibit diverse encodings in Catala; we provide quantitative details about them in Table 1, namely, the lines of Catala code (including legal specification), the number of branching constructs they contain (default terms, if-then-else, pattern matches), and the size of the compiled Python code. We particularly observe that the default terms outnumber the other branching terms, arguing in favor of their importance.

Evaluation Metrics. To evaluate the completeness of concolic tools, a standard approach is to measure the code coverage achieved. In our context, this is however hard to perform: the Catala interpreter does not provide any coverage measurements, while results on compiled Catala code, e.g., in Python, are highly unreliable. Indeed, on a small program for which we can manually determine all possible execution paths, a single Python execution reaches a line coverage of 65%, while executing all program paths barely reaches 80%. We suspect that this is due to a bloated translation of default logic into mainstream languages. Our evaluation therefore focuses on analysis times, and on the number of testcases generated. We defer a discussion of the completeness of CUTECat to Sec. 5.2.

Solver Calls and Generated Tests. We compare in Table 2 the number of solver calls performed and the number of tests generated by CUTECat. These numbers are provided both for naive executions without any optimization enabled and when all optimizations are enabled. The difference in generated tests evaluates the impact of pruning redundant exploration paths through pattern matching case folding; the difference in solver calls represents the additional impact of our optimizations targeting the SMT encoding, i.e., our handling of trivial constraints. While these optimizations lead to improvements on almost all

Benchmark	Section 132	SMIC	Family Quotient	Housing Benefits
Catala LOC	133	368	776	19655
nb(default terms)	15	17	46	251
nb(if-then-else)	3	0	24	171
nb(match)	2	0	14	139
Python LOC	343	586	1413	31826

Table 1: Statistics about the benchmarks used in the ablation study

Category	Section 132		SMIC		Family Quotient		Housing Benefits	
Optimizations	None	All	None	All	None	All	None	All
Solver Calls	41	24	138	138	13224	4142	355848	126147
Generated Tests	10	10	17	17	381	381	24995	17435

Table 2: Impact of optimizations on solver calls and generated tests

Options	Section 132		SMIC		Family Quotient		Housing Benefits	
	Total	Solve	Total	Solve	Total	Solve	Total	Solve
Standard	0.27s	0.23s	1.01s	0.85s	82.61s	69.46s	6002.03s	2262.38s
Incremental	0.02s	0.01s	0.08s	0.01s	5.21s	0.23s	4306.72s	69.88s

Table 3: Solver and total execution times, with and without incremental solving.

benchmarks (SMIC has no matches or trivial constraints, so it is not affected), their precise impact heavily differs. Avoiding calls to solver when considering trivial constraints is almost always beneficial; it reduces the number of calls up to 3.2x for the family quotient benchmark. Conversely, pattern matching folding only helps the housing benefits; other, smaller examples do not rely as much on large pattern matching. The tables below provide a more detailed analysis of the impact of individual optimizations.

Incremental Solving. We evaluate in Table 3 the gains provided by Z3's incremental mode (Sec. 4.4). To do so, we compare both the total execution times and the solver execution times when running CUTECat. Similarly to other works [51], our empirical results suggest that incremental solving is a cornerstone of efficient concolic execution; it provides more than an order of magnitude improvement to the solving times. On the smaller examples, these order-of-magnitude improvements are also observed on the total running time of the concolic engine, as the SMT solving dominates the running times. On larger programs, e.g., housing benefits, CUTECat's execution time is however dominated by running the concolic interpreter; solving constraints only represents 35% of the total time. On this example, incremental solving is 32x faster than Z3's default mode, leading to a total gain of more than 35 minutes.

Ablation Study. To evaluate the impact of each evaluation, we perform an ablation study and compare total running times for each optimization individually

Optimizations	Section 132	SMIC	Family Quotient	Housing Benefits
None	0.27s ± 0.00	1.01s ± 0.01	82.61s ± 0.55	6002.03s ± 19.46
Lazy-default	0.27s ± 0.00	1.02s ± 0.01	84.31s ± 1.15	5995.01s ± 15.21
Exception packing	0.27s ± 0.00	0.98s ± 0.01	83.41s ± 1.58	6013.71s ± 22.12
Incremental	0.02s ± 0.00	0.08s ± 0.00	5.21s ± 0.04	4306.72s ± 77.78
Trivial	0.09s ± 0.00	0.99s ± 0.01	12.62s ± 0.11	4258.93s ± 22.88
Match folding	0.21s ± 0.00	0.99s ± 0.01	46.89s ± 0.54	4045.87s ± 21.30
Frontend opts.	0.17s ± 0.00	0.99s ± 0.00	38.20s ± 0.42	3538.32s ± 22.37
All	0.02s ± 0.00	0.08s ± 0.00	4.34s ± 0.07	2843.01s ± 15.02
Speedup	1250.00%	1162.50%	1803.46%	111.12%

Table 4: Ablation study for CUTECat optimizations. Average on eight runs, with standard deviation displayed after ±.

against the bare implementation of CUTECat (without any optimization), and all optimizations enabled at once. Results are presented in Table 4. We consider the following optimizations: lazy-default and exception packing (Sec. 4.1), incremental solving (Sec. 4.4), constraint simplifications (Sec. 4.4), pattern matching case folding (Sec. 4.3), and frontend optimizations already offered by the Catala compiler. They currently consist in partial evaluation of booleans, and simplification of trivial defaults, i.e., when there are no exceptions and the default condition is a boolean value.

We observe that almost all optimizations provide noticeable performance improvements, but with variability on benchmarks likely tied to variance in their structure. For smaller programs where solving time dominates, the largest improvements are due to incremental solving.

For housing benefits however, other optimizations reduce the total time by an additional 25%. To better describe this example, we measure in Fig. 12 the evolution of the number of tests generated through time. Findings include several results previously observed: the optimizations prune the exploration tree by removing redundant cases, and incremental solving significantly improves total execution time. Additionally, we observe a linear generation rate, suggesting that concolic execution does not struggle to reach certain paths, which could have been a shortcoming of less systematic approaches, e.g., black-box fuzzing.

Notably, optimizations related to default terms, i.e., lazy default and exception packing do not impact the running times of CUTECat. This is however expected: these optimizations are only of interest when terms can return conflict values. Our benchmarks only contain mature Catala programs, corresponding to implementations of enacted laws. To the best of our knowledge, no corpus of conflict-inducing Catala examples currently exists, making a precise evaluation of these optimizations difficult. We however expect these optimizations to shine when applied during development processes, as well as to identify possible inconsistencies during the preparation of new legislation.

Soft Constraints. We evaluate the impact of using soft constraints in Table 5; the evaluation is performed with all optimizations enabled. We only consider
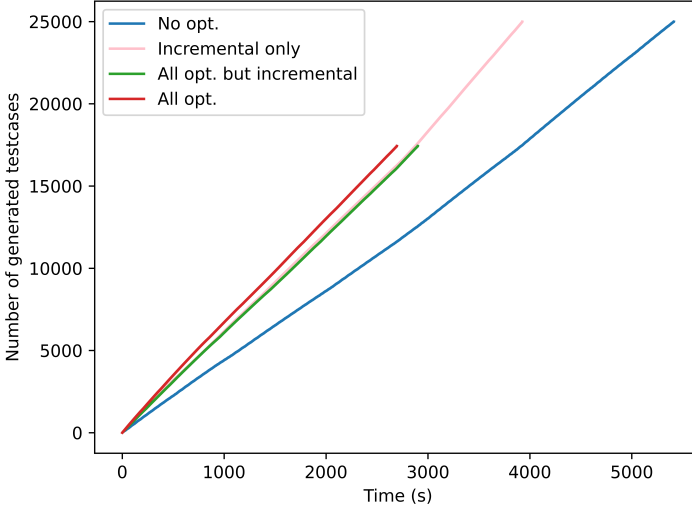
Fig. 12: Comparison of test generation rates on the housing benefits benchmark

Benchmark	Section 132	Housing Benefits
All	0.02s ± 0.00	2843.01s ± 15.02
Soft. cons.	0.03s ± 0.00	2968.80s ± 16.90
Slowdown	50.00%	4.42%
Number of tests	10	17435
% sat soft cons. (multiples of 100)	100%	87.04%
% sat soft cons. (multiples of 10)	N.A.	9.98%
% sat soft cons. (multiples of 1)	N.A.	2.80%

Table 5: Impact of soft constraints

the Section 132 and housing benefits benchmarks, as our soft constraints target monetary amounts given as inputs, which other examples do not have. On large, representative Catala examples, we observe that the computational overhead of using soft constraints is small. Additionally, the soft constraints are satisfied for almost all cases, significantly improving the usability of CUTECat. Indeed, 87.04% of the 17435 generated testcases can be solved with a soft constraint ensuring monetary amounts are multiples of 100, and additionally 9.98% and 2.80% for the multiples of 10 and units, respectively. In total, only 30/17435 generated testcases (0.17%) do not satisfy any soft constraint. Our encoding of soft constraints particularly benefits from incremental solving: they are only added to a path condition proved satisfiable by the solver – which corresponds to refining an existing solution. Furthermore, incremental solving significantly reduces solving time, minimizing the impact on CUTECat's running time of calling the solver several times on a given path. As such, we expect further extensions to soft constraints to have a minor impact on CUTECat's performance.

5.2   Detecting Errors with CUTECat

We now aim to determine whether CUTECat is effective in detecting issues in computational law implementations. Unfortunately, existing Catala codebases are mature and implementing real-world, enacted laws; they are therefore highly unlikely to exhibit such issues. To evaluate CUTECat's completeness, we therefore turn to mutation testing [26] to automatically inject errors in Catala programs, thereby simulating coding mistakes or legislative imprecisions.

We apply mutation testing to our most complex example: housing benefits. To do so, we first randomly select a default term in the program; we then either remove a chosen number of exceptions (possibly leading to unhandled cases), duplicate an exception (thus creating a conflict), or negate the default condition of a default term (possibly leading to unhandled cases). We follow this process to generate 20 new programs, which we manually inspect to confirm that mutations introduced issues. In particular, we must ensure that the default term considered is reachable, and that assumptions and conditions in other parts of the code do not preclude the unhandled cases, or the execution of the conflicting exceptions.

CUTECat successfully identifies all 20 issues, providing concrete inputs to identify problematic cases. On average, CUTECat requires 0.55 seconds to reach the injected bug; this execution time leads us to believe that our tool would be particularly suitable as part of continuous integration workflows, but also to evaluate intended amendments to laws.

5.3   Case Study: Rental Housing Benefits

Our evaluation so far focused on a constrained case of the rental housing benefits, considering only households located in mainland France and with less than 10 children. We chose to enforce these restrictions due to the multiple executions required to evaluate different CUTECat settings, which already required more than one hour each. In this section, we now explore the general, unconstrained case of the rental housing benefits. With all optimizations enabled, CUTECat is able to analyze this program in 6h37m, generating 186390 testcases. This process generates 1338575 solver calls; the total solver time is 366s.

Interestingly, CUTECat's analysis led to the discovery of a conflict error in the existing implementation. The housing benefits are defined for various cases, including when the flat is shared with other roommates, or when a single bedroom is rented. However, an interpretation conflict can happen for people claiming rental housing benefits if they are sharing a bedroom with roommates. This conflict was previously manually discovered by Merigoux [35] during their implementation of the housing benefits. Despite querying relevant public administrations, this case was deemed unlikely to happen, and no legal interpretation was provided to resolve this ambiguity. No other errors have been found by CUTECat on the rental housing benefits.

A manual inspection of the generated testcases revealed that the current input space for the housing benefits is under-constrained. For example, some testcases correspond to households located overseas and in housing zone 3 – but

to the best of our knowledge, all overseas territories are defined to be in housing zone 2. Such cases exemplify where expert knowledge from lawyers would be beneficial: provided with indications about which inputs are well-formed, programmers can then restrict CUTECat's search space through program assertions, therefore speeding up the analysis.

Overhead of Concolic Execution. We now compare the overhead of the concolic execution of CUTECat with the interpretation times of all the generated testcases. The reference Catala interpreter evaluates the 186390 generated tests in 1h29, meaning CUTECat has a 4.5x overhead. While large, this overhead is not unexpected: the concolic interpreter instruments each expression evaluation to collect symbolic constraints, and performs repeated calls to the SMT solver. Prior work on concolic execution observed similar trends: for example, Yun et al. [58] measure that KLEE [8] has up to a three order-of-magnitude overhead, while our experimental results are in line with approaches focusing on reducing this overhead [12, 46]. This leads us to believe that CUTECat's scalability is comparable to concolic tools in other languages; we nevertheless intend to perform a deeper profiling of our interpreter to identify potential bottlenecks. This comparison also allows us to sanity-check our implementation with respect to Catala's semantics: we confirmed that CUTECat and the Catala reference interpreter agreed on all generated testcases. Furthermore, we can use those same testcases to check whether all Catala backends have the same behavior on the case study.

6 Related Work

Concolic Execution. Concolic execution was introduced by the seminal works of Godefroid et al. [21] and Sen et al. [51] almost two decades ago. Since then, it has been applied to various programming languages [8, 9, 31, 34, 50]. We refer the reader to the survey of Baldoni et al. [1] for an extensive coverage of concolic and symbolic execution. As Catala has different features compared to traditional, imperative programming languages, most of the advanced techniques mentioned in the survey (memory encoding, loop summarization) are not relevant for our work. Similarly, human-readability of generated tests is usually not an objective of traditional concolic execution engines. However, Baldoni et al. [1] mention that "the way high-level switch statements are compiled can significantly affect the performance of path exploration", echoing in an imperative setting the need for an encoding similar to the pattern-matching case folding presented in Sec. 4.3. Giantsios et al. [19] support pattern matching through a compilation to a decision tree, which is the approach used to generate machine code [33]. In Sec. 4.3, we chose to implement a simpler approach as Catala currently supports matching against a single sum type. As we have mentioned in Sec. 5.2, our mutation approach is lacking information to be reliable in user-constrained cases, which required us to manually inspect mutated programs. To create an analysis-oriented benchmarking suite for Catala programs, an interesting avenue would be to rely on the program generation techniques of Vikram et al. [56].

Formal Methods for the Law. Several approaches were previously proposed to reason about default logic. Risch [48] and Cassano et al. [11] both define tableaux-based approaches [25] to reason about propositional formulas expressed in default logic, while Schaub [49] and Linke and Schaub [30] propose query answering methodologies for default logic propositions. By focusing on the underlying logic, these works differ from our approach tailored to program verification, which emphasizes testcase generation and requires reasoning about numerical expressions and program constructs such as structures or enumerations. In their compiler for the French tax code, Merigoux et al. [38] have relied on coverage-guided fuzzing with AFL [60] to improve the quality of a pre-existing test-suite. Their approach however only generated 275 minimized testcases from a first generation of 30,000 cases. In contrast, CUTECat's concolic-based approach allows to generate cases exploring different execution paths, while reaching rare ambiguous or unhandled situations. Monat et al. [41] and de Almeida Borges et al. [17] focus on formalizing date-duration arithmetic and semantics of UTC time respectively, with the former also analyzing date-related ambiguities in legal computations. Our work currently only provides limited support for date operations, but could build upon these formalizations to define suitable symbolic encodings.

7 Conclusion and Future Work

Legal expert systems implementing computational laws are pervasive, and faithfully translating the law into code is error-prone. To alleviate this issue, we proposed CUTECat, a concolic execution tool targeting computational law implementations in the Catala language. CUTECat relies on a novel concolic semantics for default terms, and is equipped with several optimizations improving its scalability and usability by lawyers.

CUTECat scales to Catala codebases implementing real-world French and US laws, generating hundreds of thousands of unique testcases in less than seven hours of CPU time. We believe this paper takes a step towards the use of formal methods during legislative processes. Our future work includes tailoring CUTECat's pipeline and interface for lawyers through interdisciplinary experimentation. We also plan to discuss additional usecases for CUTECat with law experts; namely, we believe CUTECat could be used to improve state-of-the-art computer-assisted tax teaching techniques [29].

References

[1] Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018), https://doi.org/10.1145/3182657

[2] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: Cvc5: A versatile and industrial-strength SMT solver. In: Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Lecture Notes in Computer Science, vol. 13243, pp. 415–442, Springer (2022), https://doi.org/10.1007/978-3-030-99524-9_24

[3] Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB) (2016), URL https://www.SMT-LIB.org

[4] Barrett, C., Kroening, D., Melham, T.: Problem solving for the 21st century: Efficient solvers for satisfiability modulo theories. Tech. Rep. 3, London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering (Jun 2014), URL http://theory.stanford.edu/~barrett/pubs/BKM14.pdf, knowledge Transfer Report

[5] Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—a formal system for testing and debugging programs by symbolic execution. ACM SigPlan Notices **10**(6), 234–245 (1975), https://doi.org/10.1145/390016.808445

[6] Brewka, G., Eiter, T.: Prioritizing default logic. In: Intellectics and Computational Logic: Papers in Honor of Wolfgang Bibel, pp. 27–45, Springer (2000)

[7] Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: Proceedings of the ACM European Conference on Computer Systems (EuroSys) (2011), https://doi.org/10.1145/1966445.1966463

[8] Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2008), URL https://dl.acm.org/doi/10.5555/1855741.1855756

[9] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) (2008), https://doi.org/10.1145/1455518.1455522

[10] Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Communications of the ACM **56**(2), 82–90 (2013), https://doi.org/10.1145/2408776.2408795

[11] Cassano, V., Fervari, R., Hoffmann, G., Areces, C., Castro, P.F.: A tableaux calculus for default intuitionistic logic. In: Proceedings of the International

Conference on Automated Deduction (CADE) (2019), https://doi.org/10.1007/978-3-030-29436-6_10

[12] Chen, J., Han, W., Yin, M., Zeng, H., Song, C., Lee, B., Yin, H., Shin, I.: SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In: USENIX Security Symposium, pp. 2531–2548, USENIX Association (2022), URL https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju

[13] Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2011), https://doi.org/10.1145/1950365.1950396

[14] Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on software engineering (3), 215–222 (1976), https://doi.org/10.1109/TSE.1976.233817

[15] Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability modulo Theories, Oxford, United Kingdom (Jul 2018), URL https://inria.hal.science/hal-01960203

[16] David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In: Proceedings of the IEEE International Conference on Software Analysis, Evoluation, and Reengineering (SANER) (2016), https://doi.org/10.1109/SANER.2016.43

[17] de Almeida Borges, A., Bedmar, M.G., Rodríguez, J.J.C., Reyes, E.H., Buñuel, J.C., Joosten, J.J.: UTC time, formally verified. In: Proceedings of the International Conference on Certified Programs and Proofs (CPP), pp. 2–13, ACM (2024), https://doi.org/10.1145/3580430

[18] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2008), https://doi.org/10.1007/978-3-540-78800-3_24

[19] Giantsios, A., Papaspyrou, N., Sagonas, K.: Concolic testing for functional languages. Science of Computer Programming **147**, 109–134 (2017), https://doi.org/10.1016/j.scico.2017.04.008

[20] Godefroid, P.: Compositional dynamic test generation. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL), pp. 47–54, POPL '07, Association for Computing Machinery, New York, NY, USA (2007), https://doi.org/10.1145/1190216.1190226

[21] Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), pp. 213–223, ACM (2005), https://doi.org/10.1145/1065010.1065036

[22] Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Sympo-

sium (NDSS) (2008), URL https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[23] Goutagny, P., Fromherz, A., Monat, R.: CUTECat: Concolic Execution for Computational Law (Artifact) (Jan 2025), https://doi.org/10.5281/zenodo.14677860

[24] Howden, W.: Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering **SE-3**(4), 266–278 (1977), https://doi.org/10.1109/TSE.1977.231144

[25] Howson, C.: Logic with Trees: An Introduction to Symbolic Logic. Routledge (2005)

[26] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011), https://doi.org/10.1109/TSE.2010.62

[27] King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976), https://doi.org/10.1145/360248.360252

[28] Lawsky, S.B.: A logic for statutes. Fla. Tax Rev. **21**, 60 (2017), https://doi.org/10.2139/ssrn.3088206

[29] Lawsky, S.B.: Teaching algorithms and algorithms for teaching (2021), https://doi.org/10.5744/ftr.2021.2004

[30] Linke, T., Schaub, T.: Lemma handling in default logic theorem provers. In: European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty, pp. 285–292, Springer (1995), https://doi.org/10.1007/3-540-60112-0_33

[31] Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: jDart: A dynamic symbolic analysis framework. In: Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2016), https://doi.org/10.1007/978-3-662-49674-9_26

[32] Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of the International Conference on Software Engineering (ICSE) (2007), https://doi.org/10.1109/ICSE.2007.41

[33] Maranget, L.: Compiling pattern matching to good decision trees. In: Proceedings of the ACM SIGPLAN Workshop on ML, pp. 35–46, ACM (2008), https://doi.org/10.1145/1411304.1411311

[34] Marques, F., Fragoso Santos, J., Santos, N., Adão, P.: Concolic execution for WebAssembly. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Leibniz International Proceedings in Informatics (Lipics), vol. 222, pp. 11:1–11:29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022), https://doi.org/10.4230/LIPIcs.ECOOP.2022.11

[35] Merigoux, D.: Experience report: Implementing a real-world, medium-sized program derived from a legislative specification. Tech. rep. (Jan 2023), URL https://inria.hal.science/hal-03933574

[36] Merigoux, D., Barnes, J., Botbol, V., Estep, S., Gerrior, M., Gesbert, L., Rolley, E., Salek, A., Scott, E., Török, E.: English tutorial for catala developers (2024), URL https://catala-lang.org/en/examples/tutorial

[37] Merigoux, D., Chataing, N., Protzenko, J.: Catala: A programming language for the law. Proceedings of the International Conference on Functional Programming (ICFP) **5**(ICFP), 77:1–77:29 (2021), https://doi.org/10.1145/3473582

[38] Merigoux, D., Monat, R., Protzenko, J.: A modern compiler for the French tax code. In: Proceedings of the International Conference on Compiler Construction (CC), pp. 71–82, CC 2021, ACM, New York, NY, USA (2021), https://doi.org/10.1145/3446804.3446850

[39] Micinski, K., Fetter-Degges, J., Jeon, J., Foster, J.S., Clarkson, M.R.: Checking interaction-based declassification policies for android using symbolic execution. In: Proceedings of the European Symposium on Research in Computer Security (ESORICS) (2015), https://doi.org/10.1007/978-3-319-24177-7_26

[40] Mockler, P.: The Phoenix Pay Problem: Working toward a Solution. Senate Canada (2018), URL https://publications.gc.ca/pub?id=9.859900&sl=0

[41] Monat, R., Fromherz, A., Merigoux, D.: Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law. In: Proceedings of the European Conference on Programming Languages and Systems (ESOP), Lecture Notes in Computer Science, vol. 14577, pp. 421–450, Springer, Cham (2024), https://doi.org/10.1007/978-3-031-57267-8_16

[42] Monin, J.: Louvois, le logiciel qui a mis l'armée à terre (2018), URL https://www.radiofrance.fr/franceinter/podcasts/secrets-d-info/louvois-le-logiciel-qui-a-mis-l-armee-a-terre-8752880

[43] Nguyễn, P.C., Van Horn, D.: Relatively complete counterexamples for higher-order programs. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), pp. 446–456, PLDI '15, Association for Computing Machinery, New York, NY, USA (2015), https://doi.org/10.1145/2737924.2737971

[44] Noller, Y., Kersten, R., Păsăreanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (2018), https://doi.org/10.1145/3213846.3213868

[45] Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 179–180, ASE '10, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1858996.1859035

[46] Poeplau, S., Francillon, A.: Symbolic execution with SymCC: Don't interpret, compile! In: Proceedings of the 29th USENIX Conference on Security Symposium, pp. 181–198, SEC'20, USENIX Association, USA (2020), URL https://dl.acm.org/doi/10.5555/3489212.3489223

[47] Redden, J., Brand, J., Sander, I., Warne, H., Grant, A., White, D.: Automating public services: Learning from cancelled systems. Cardiff, UK: Data Justice Lab, Cardiff University (2022),

URL   https://carnegieuk.org/publication/automating-public-services-learning-from-cancelled-systems/

[48] Risch, V.: Analytic tableaux for default logics. Journal of Applied Non-Classical Logics **6**(1), 71–88 (1996), https://doi.org/10.1080/11663081.1996.10510867

[49] Schaub, T.: A new methodology for query answering in default logics via structure-oriented theorem proving. Journal of Automated Reasoning **15**, 95–165 (1995), https://doi.org/10.1007/BF00881832

[50] Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Proceedings of the International Conference on Computer Aided Verification (CAV), Lecture Notes in Computer Science, vol. 4144, pp. 419–423, Springer (2006), https://doi.org/10.1007/11817963_38

[51] Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/SIGSOFT FSE, pp. 263–272, ACM (2005), https://doi.org/10.1145/1081706.1081750

[52] Sivaramakrishnan, KC., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto ocaml. In: Proceedings of the International Conference on Functional Programming (ICFP) (2020), https://doi.org/10.1145/3408995

[53] Staats, M., Păsăreanu, C.: Parallel symbolic execution for structural test generation. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (2010), https://doi.org/10.1145/1831708.1831732

[54] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2016), https://doi.org/10.14722/ndss.2016.23368

[55] Tobin-Hochstadt, S., Van Horn, D.: Higher-order symbolic execution via contracts. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 537–554, OOPSLA '12, Association for Computing Machinery, New York, NY, USA (2012), https://doi.org/10.1145/2384616.2384655

[56] Vikram, V., Padhye, R., Sen, K.: Growing a Test Corpus with Bonsai Fuzzing. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 723–735, IEEE (May 2021), https://doi.org/10.1109/ICSE43902.2021.00072

[57] Wikipedia contributors: Code of Ur-Nammu — Wikipedia, the free encyclopedia (2024), URL https://en.wikipedia.org/w/index.php?title=Code_of_Ur-Nammu&oldid=1243860044

[58] Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In: USENIX Security Symposium, pp. 745–761, USENIX Association (2018), URL https://dl.acm.org/doi/10.5555/3277203.3277260

[59] Z3 Guide: Soft constraints (2024), URL https://microsoft.github.io/z3guide/docs/optimization/softconstraints

[60] Zalewski, M.: American fuzzy lop. GitHub (Jul 2020), URL https://github.com/google/AFL