

SPEC2CODE: Mapping Protocol Specification to Function-Level Code Implementation

Yuekun Wang, Lili Quan, Xiaofei Xie,
Junjie Wang, Jianjun Chen
ASE 2025

導入

背景

○プログラムが設計図通り

⇒検証が重要

課題

プログラムと設計図の対応関係の把握が必要

⇒膨大な時間と労力

AIの可能性と課題

成長した大規模言語モデルなら、深い意味を理解して対応関係を把握できそう

- ・ 計算コスト
- ・ 文脈不足による誤検知
- ・ 多対多

新しいフレームワーク『SPEC2CODE』

関連研究

トレーサビリティ・リンク回復 (TLR)

- 多様なアプローチによる精度向上
- 語彙不一致への対応
- ×ファイルレベルのトレーサビリティに制限
- ×前処理された成果物に依存

コード検索

- シンプルなキーワード検索
- 意味的な検索の実現
- ×機能の検索に特化している
- ×キーワードへの過剰な依存
- ×1対1が前提

⇒

- ・生データから直接、関数レベルの細かい紐づけ
- ・多対多の紐づけに対応

SPEC2CODE

動作

1. RFC文書とソースコードを入力
2. 情報の抽出
3. 候補の絞り込み
4. マッピング
5. 出力

情報抽出

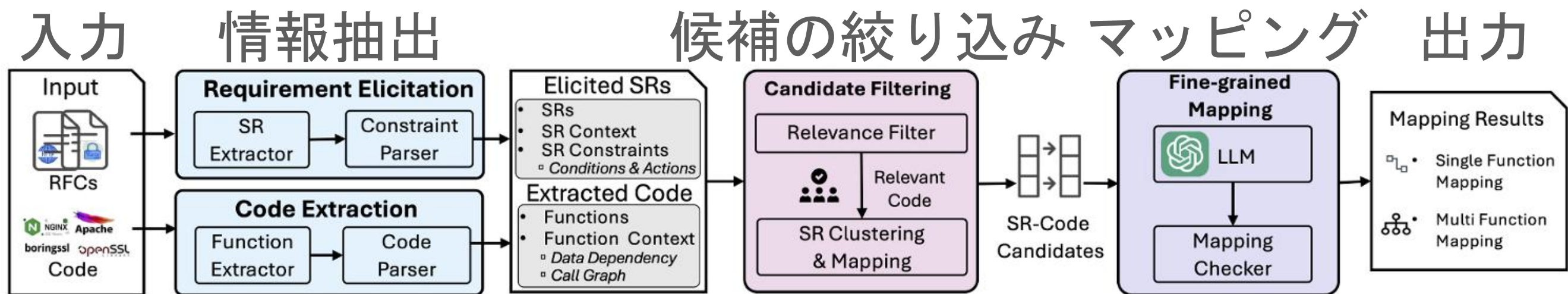


Fig. 1: Overview of SPEC2CODE

問題定義

定義

C : ソースコード全体

S : 仕様要件の集合

s : 仕様要件

f : 関数

① 関連性

- ・ 完全or部分的 : f が s のすべての側面をカバーしているか、部分的か
- ・ 正しいor誤り : f が s の意味と一致しているかどうか

② 一貫性と不一致

- ・ 一貫 : 完全かつ正確
- ・ 不一致 : どちらか片方が部分的や誤りを含む

③ 仕様からコードへのマッピング

目標 : マッピング $M \subseteq S \times P(C)$

各ペア $(s, C_i) \in M$

実験概要

リサーチクエスチョン（RQ）

RQ1：SPEC2CODEによるマッピングにはどんな特徴がある？

RQ2：SPEC2CODEのパフォーマンスは、既存手法と比較して優れている？

RQ3：SPEC2CODEの各機能は全体性能にどう貢献している？

RQ4：生成されたマッピングを使って、未知の新たなバグは発見できる？

使用されたデータセット

①ベンチマークデータセット

- ・ 概要：HTTP/1.1の使用要件とその実装である「Apache HTTPD」プロジェクトのソースコードの対応関係を手作業で検証正解データとしてまとめたもの
- ・ 使用箇所：SPEC2CODEや既存のベースライン手法のテスト問題

②不整合バグデータセット

- ・ 概要：過去に報告された既知のソフトウェアバグを集めたデータセット
NVDやGitHubのIssueなどから収集
- ・ 使用箇所：実際のソフトウェアにおける過去のバグの箇所の発見可能性検証

実験結果（RQ1）

RQ1：SPEC2CODEによるマッピングにはどんな特徴がある？

TABLE II: Spec-Code Mapping and Coverage Statistics.

Protocol	Impl	#Function	LOC	#Mapping (SFM/MFM)	#SR Covered	MUST	SHOULD	MAY
HTTP	HTTPD	7,491	942K	641 (573/68)	263/327 (0.90)	142/171 (0.94)	75/95 (0.88)	46/61 (0.83)
	Nginx	2,819	168K	332 (302/30)	210/327 (0.92)	117/171 (0.95)	58/95 (0.89)	35/61 (0.85)
TLS	OpenSSL	3,283	648K	397 (368/29)	247/410 (0.88)	177/306 (0.92)	40/60 (0.80)	30/44 (0.72)
	BoringSSL	2,285	197K	346 (322/24)	212/410 (0.89)	152/306 (0.93)	35/60 (0.82)	25/44 (0.75)
BFD	FRRouting	3,072	686K	188 (170/18)	145/170 (0.89)	87/100 (0.93)	38/45 (0.86)	20/25 (0.80)
	BIRD	218	36K	92 (84/8)	84/170 (0.87)	46/100 (0.91)	24/45 (0.83)	14/25 (0.78)

SFM：1つの関数 網羅率（正解率）

MFM：複数の関数

結果

- ・実装間でマッピングに大きなばらつき
- ・要件レベルの性質
 - MUST：見つけやすい
 - SHOULD/MAY：そもそもコードが存在しない可能性あり
 - MUST NOT：AI「禁止するためのコードを探す」, 正解「何もしないこと」

実験結果 (RQ 2)

RQ2 : SPEC2CODEのパフォーマンスは、既存手法と比較して優れている？

実験内容

- (1)ベンチマークに基づく評価
- (2)手動検査
- (3)既知の不整合バグのカバレッジ

結果

- (1)(2)他と比べて圧倒的スコア
- (3)過去に本当に合ったバグのデータに対してバグをすべて発見

TABLE III: Performance Comparison of SPEC2CODE and Baselines on Benchmark

Tool	Single-Function				Multi-Function			
	SR@1	Precision	Recall	F1	SR@ G _s	Precision	Recall	F1
DeepCS	0.54	0.55	0.58	0.56	0.17	0.17	0.17	0.17
deGraphCS	0.55	0.57	0.59	0.58	0.21	0.21	0.21	0.21
LiSSA	0.62	0.63	0.34	0.44	0.17	0.17	0.17	0.17
FTLR	0.30	0.32	0.55	0.40	0.08	0.08	0.08	0.08
Spec2Code	0.95	0.94	0.98	0.96	0.96	0.92	0.96	0.94

TABLE IV: The Result of Manual Inspection for Single-Function (SFM) and Multi-Function (MFM) Mappings

Impl.	Spec2Code		LiSSA		FTLR		DeepCS		deGraphCS	
	SFM	MFM	SFM	MFM	SFM	MFM	SFM	MFM	SFM	MFM
HTTPD	0.90	0.80	0.54	0.10	0.46	0.10	0.48	0.10	0.52	0.10
Nginx	0.92	0.90	0.50	0	0.48	0	0.44	0	0.56	0
OpenSSL	0.88	0.90	0.44	0.10	0.42	0.10	0.46	0	0.48	0.10
BoringSSL	0.90	0.80	0.50	0	0.38	0.10	0.42	0	0.42	0
FRR	0.86	0.80	0.40	0	0.36	0	0.40	0	0.38	0.10
BIRD	0.84	0.70	0.36	0.10	0.32	0	0.30	0	0.40	0
Average	0.88	0.82	0.46	0.05	0.40	0.05	0.42	0.02	0.46	0.05

実験結果（RQ3）

RQ3：SPEC2CODEの各機能は全体性能にどう貢献している？

実験内容

(1) モジュール削除によるシステムレベルのアブレーション

結果

関連性フィルタリング：効率性

SRクラスタリング：効率性

関数レベルのコンテキスト：精度

TABLE V: Performance comparison of SPEC2CODE and its variants on the HTTP benchmark dataset.

Tool	Precision	Recall	F1	# LLM Calls
SPEC2CODE	0.94	0.98	0.96	4,023
SPEC2CODE_NRF	0.91	0.98	0.94	6,057
SPEC2CODE_NCP	0.87	1.00	0.93	34,695
SPEC2CODE_NFC	0.89	0.95	0.92	4,023
PURELLM	0.88	0.96	0.92	70,959

NRF=関連性フィルタなし

NCP=SRクラスタリングなし

NFC=関数レベルのコンテキストなし

実験結果（RQ 3）

RQ3：SPEC2CODEの各機能は全体性能にどう貢献している？

実験内容

(2)LLM主導の各ステップ（制約パーサー,関連性フィルタ,意味分類,マッピング検証）の本質的な信頼性を評価する検証

結果

- すべてのステップで高い信頼性
- ハルシネーションの影響も無視できる程度

TABLE VI: Component-level validation of LLM-driven steps in SPEC2CODE.

Component	Metric (#Sample)	Result
Constraint Parser	Accuracy (50)	0.98
Relevance Filter	Recall (100)	1.00
Semantic Classification	Accuracy (50)	0.98
Mapping Verification	P / R / F1 (100)	0.94 / 1.00 / 0.97

実験結果（RQ4）

RQ4：生成されたマッピングを使って、未知の新たなバグは発見できる？

実験内容

- ① マッピング
- ② GPT-o3-miniとDeepSeek-R1によって不整合検出

結果

- ・ 15/56の真の不整合（精度約27%）
 - ・ 11/15は未知の不整合
 - ・ 5/11はプロジェクトの開発者によって承認
- ⇒バグ発見を支援するうえで実用的な有用性？

考察

学び

- ・ マッピング品質の向上
 - MUST NOTや外部ライブラリに委譲された要件の追跡の困難さ
- ・ 不整合検出におけるLLMの有望性と課題
 - 未知の不整合を発見⇒LLMベースの推論の可能性
 - 偽陽性率が高い

コストとスケーラビリティ

- ・ 大規模なプロトコルコードベースに対しても実行可能かつ安価

TABLE VII: Runtime and cost of applying Spec2Code across implementations.

	HTTPD	Nginx	OpenSSL	BoringSSL	FRRouting	BIRD
Runtime (H)	3.1	1.1	1.3	0.9	1.4	0.4
Cost (USD)	20.3	7.5	8.8	6.1	9.6	2.5

妥当性への脅威

- ・ 内部的：2名の研究者が独立してタスク実行
- ・ 外部的：評価がプロトコルの実装のみに焦点⇒汎用性の制限の可能性

所感

学び

- ・ 自然言語の構造化の有効性
 - 一曖昧な自然言語で書かれた要件を「トリガー」「アクション」に分解
- ・ 正解ラベルの付け方
 - 一AIが提案してきたマッピングが妥当かどうかを正解ラベル付け
- ・ MUST NOTの難しさ
- ・ クラスタリングによるコストの抑制の重要性