*O*ptimized data structure and query system f*o*r static c*o*de analysis

Quentin Dauprat

Normandie Université

UNIVERSITÉ CAEN NORMANDIE

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité **INFORMATIQUE**

Préparée au sein de l'**Université de Caen Normandie**

Optimized data structure and query system for static code analysis

Présentée et soutenue par

DAUPRAT QUENTIN

Thèse soutenue le 05/11/2025

devant le jury composé de :

M. DORBEC PAUL	Professeur des universités - Université de Caen	Directeur de thèse
M. SINGHOFF FRANCK	Professeur des universités - Universite Bret. Occidentale Ubo	Président du jury
M. RICHARD GAETAN	Maître de conférences - UCN - Université de Caen Normandie	Co-encadrant
M. MINÉ ANTOINE	Professeur des universités - Sorbonne Université	Membre du jury
MME PEKERGIN NIHAL	Professeur des universités - UNIVERSITE PARIS 12 VAL DE MARNE	Membre du jury
MME ROSEN JEAN-PIERRE	Chercheur - ADALOG	Membre du jury
M. CHALLOUX EMMANUEL	Professeur émérite - Sorbonne Université	Rapporteur du jury

Thèse dirigée par **DORBEC PAUL** (Groupe de recherche en informatique, image et instrumentation de Caen)

MIIS

GREYC
Laboratoire de recherche en sciences du numérique

Keywords: ada language, graph databases, neo4j, pattern matching, scalability, static code analysis, coding rule verification

Mots clés : langage ada, bases de données orientées graphe, neo4j, filtrage par motif, passage à l'échelle, analyse statique de code, vérification de règles de codage

This thesis has been prepared at the following research units.

GREYC

6 Boulevard du Maréchal Juin
Bâtiment Sciences 3
CS 14032
14032 CAEN cedex 5
France

@ contact@greyc.fr
https://www.greyc.fr

Adalog

2 rue Mozart
92110 Clichy
France

@ contact@adalog.fr
https://www.adalog.fr

Novasys Ingenierie

2 rue Mozart
92110 Clichy
France

@ contacts2@pactenovation.fr
https://www.novasys-ingenierie.com/

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty

---

Donald Knuth

Quality means doing it right when no one is looking

Henry Ford

Optimized data structure and query system for static code analysis

Abstract

Static code analysis encompasses various techniques for improving software quality and security. In this research, we focus exclusively on one important aspect: the verification of coding rules. Conventional approaches for coding rule verification face challenges in efficiently analyzing large, complex codebases. We thus explore the potential of graph databases to enhance the performance of this specific static analysis task.

We propose a graph-based methodology that represents source code as rich property graphs, enabling intuitive modeling of syntax, semantics, and behavior specifically for coding rule verification. We parse the codebase and populate it into a graph database. Then, we evaluate coding rules through graph traversals expressed in the Cypher query language, converting traditional rule checks into optimized graph patterns.

We implemented this approach in a prototype tool, entitled Cogralys, for Ada and evaluated it on real-world benchmarks. Our experiments demonstrate significant runtime improvements in coding rule verification: Cogralys completes analyses 6.3 times faster than AdaControl and 17.6 times faster than GNATcheck. For specific rule categories, we achieved even greater improvements– up to 195 times faster for local rules compared to traditional analyzers. These results confirm graph databases' capacity to accelerate coding rule verification through optimized data structures and parallel query processing.

However, overheads introduced by database population should be considered. We found the technique is best suited for sizable, frequently analyzed code. While showing significant promise for coding rule verification, more research is needed to address language support, integration with developers' workflows, and queries for more complex rules.

Overall, in this thesis we deliver a practical graph-based framework for coding rule verification while presenting the advantages, trade-offs and future opportunities of leveraging graph technologies for efficient, scalable verification of coding standards.

Keywords: ada language, graph databases, neo4j, pattern matching, scalability, static code analysis, coding rule verification

GREYC

6 Boulevard du Maréchal Juin – Bâtiment Sciences 3 – CS 14032 – 14032 CAEN cedex 5 – France

Structures de données et système de requêtes optimisés pour l'analyse statique de code

Résumé

L'analyse statique de code englobe diverses techniques pour améliorer la qualité et la sécurité des logiciels. Dans notre recherche, nous nous concentrons exclusivement sur un aspect important : la vérification des règles de codage. Nous avons identifié que les approches conventionnelles pour la vérification des règles de codage peinent à analyser efficacement des bases de code volumineuses et complexes. Nous explorons le potentiel des bases de données orientées graphe pour améliorer les performances de cette tâche spécifique d'analyse statique.

Nous proposons une méthodologie basée sur les graphes pour représenter le code source sous forme de graphe de propriétés, permettant une modélisation intuitive de la syntaxe, de la sémantique et du comportement spécifiquement pour la vérification des règles de codage. Nous analysons la base de code et l'intégrons dans une base de données orientée graphe. Nous évaluons ensuite les règles de codage par des traversées de graphe exprimées en langage de requête Cypher, convertissant les vérifications traditionnelles en motifs de graphe optimisés.

Nous avons implémenté cette approche par l'intermédiaire d'un prototype, appelé Cogralys, pour le langage Ada et l'avons évaluée sur des benchmarks du monde réel. Nos expériences démontrent des améliorations significatives en temps d'exécution pour la vérification des règles de codage : Cogralys effectue les analyses 6,3 fois plus rapidement qu'AdaControl et 17,6 fois plus rapidement que GNATcheck. Pour certaines catégories de règles, nous avons obtenu des améliorations encore plus importantes – jusqu'à 195 fois plus rapide pour les règles locales par rapport aux analyseurs traditionnels. Ces résultats confirment la capacité des bases de données graphe à accélérer la vérification des règles de codage grâce à des structures de données optimisées et à un traitement parallèle des requêtes.

Cependant, nous reconnaissons que les surcharges introduites par la population de la base de données sont à prendre en compte. Nous avons constaté que la technique est mieux adaptée pour du code volumineux et fréquemment analysé. Bien que prometteuse pour la vérification des règles de codage, nous identifions que des recherches supplémentaires sont nécessaires pour traiter la prise en charge d'autres langages, l'intégration dans les flux de développement et les requêtes pour des règles plus complexes.

Globalement, dans cette thèse nous proposons un cadre basé sur les graphes pour la vérification des règles de codage tout en présentant les avantages, les inconvénients et les opportunités futures de l'utilisation des technologies graphes pour une vérification efficace et évolutive des standards de codage.

Mots clés : langage ada, bases de données orientées graphe, neo4j, filtrage par motif, passage à l'échelle, analyse statique de code, vérification de règles de codage

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my superior, Jean-Pierre Rosen, CEO of Adalog, for accepting me into his internship program and enabling me to continue pursuing doctoral studies under his guidance. His vision and leadership were instrumental in shaping this research. I am also deeply grateful to Laurent Gouzenes, the scientific director of the company, for his valuable guidance and support throughout my research journey. I must extend my profound thanks to Novasys Ingenierie, whose generous financial support made this research possible.

I would also like to sincerely thank my thesis supervisor, Professor Paul Dorbec, for his invaluable advice and support throughout this endeavor. His insights and direction were crucial to the completion of this work.

In addition, I am extremely thankful to my co-supervisor, Dr. Gaetan Richard, for all the time and effort he dedicated to providing thoughtful feedback that helped refine and enhance the present research. This thesis has benefitted tremendously from his input.

My heartfelt appreciation goes out to my family, for their endless love and encouragement which inspired me to stay determined through all the ups and downs of this challenging but rewarding PhD journey. I could not have done this without them.

I want to acknowledge as well the many friends, colleagues, collaborators, and institutions who contributed directly or indirectly to making this research possible.

Finally, I am grateful to have had this opportunity to conduct impactful research alongside many gifted minds. The experience has been truly rewarding both personally

and professionally.

Résumé substantiel en français

Introduction et contexte

La fiabilité du logiciel constitue aujourd'hui un enjeu stratégique dans de nombreux secteurs industriels à haute criticité, tels que l'aéronautique, le spatial, le ferroviaire, le domaine médical, la défense ou encore l'énergie. Dans ces environnements, le logiciel embarqué pilote des fonctions vitales, dont la défaillance peut entraîner des conséquences humaines, matérielles et économiques majeures. L'histoire récente a été marquée par plusieurs accidents emblématiques : le système de radiothérapie Therac-25, dont des défauts logiciels ont causé plusieurs décès ; l'explosion de la fusée Ariane 5 lors de son vol inaugural, imputée à une erreur de conversion numérique ; ou encore les crashs du Boeing 737 MAX, où des erreurs logicielles dans le système MCAS ont provoqué la perte de deux appareils et de centaines de vies humaines. Ces événements soulignent la nécessité absolue de maîtriser la qualité du logiciel tout au long de son cycle de vie.

Dans ce contexte, les logiciels embarqués sont soumis à des exigences de sûreté et de certification particulièrement strictes, imposées par des normes internationales telles que DO-178C (aéronautique), ISO 26262 (automobile), EN 50128 (ferroviaire), ou encore CEI 61508 (industrie). Ces référentiels exigent la mise en œuvre de pratiques de développement rigoureuses, incluant la vérification systématique du respect de règles de codage précises, afin de prévenir l'introduction de défauts et de faciliter la maintenance.

Le langage Ada occupe une place centrale dans ce paysage. Conçu dès l'origine pour répondre aux besoins de la programmation de systèmes critiques, Ada se distingue par

son typage fort, sa gestion native de la concurrence, ses mécanismes de modularité et d'abstraction. Il est largement adopté dans les secteurs où la fiabilité et la maintenabilité du logiciel sont des exigences non négociables.

L'évolution des pratiques industrielles, la croissance constante de la taille des bases de code, la multiplication des intervenants et l'allongement du cycle de vie des logiciels rendent désormais la vérification manuelle des règles de codage inenvisageable à grande échelle. Les outils automatiques de vérification statique sont devenus indispensables pour garantir la conformité, réduire les coûts de revue, accélérer la mise sur le marché et fiabiliser les livrables dans un contexte de complexité croissante.

Problématique et objectifs

Malgré les avancées significatives des outils de vérification statique, les approches traditionnelles fondées sur l'analyse des Abstract Syntax Trees (ASTs) atteignent aujourd'hui leurs limites face à l'ampleur et à la complexité croissante des bases de code industrielles. Les outils de référence pour Ada, tels qu'AdaControl ou GNATcheck, s'appuient principalement sur une analyse structurée du code source via l'AST, mais se heurtent à plusieurs obstacles majeurs :

- **Dispersion de l'information syntaxique et sémantique** : dans les grands projets, les éléments nécessaires à la vérification d'une règle sont souvent répartis sur de multiples fichiers, packages et unités de compilation, rendant leur agrégation coûteuse et peu scalable.
- **Difficulté à exprimer des règles complexes** : certaines règles de codage requièrent de corréler des informations issues de différents niveaux de l'AST, voire de parcourir des relations de dépendance, d'appel ou d'héritage, difficiles à exprimer avec des outils conçus pour des vérifications locales.
- **Problèmes de performance et de passage à l'échelle** : l'analyse de très grands ensembles de code peut entraîner des temps de traitement prohibitifs, notamment

lorsque l'outil doit recharger ou reconstituer à la volée des arbres syntaxiques volumineux et interdépendants.

- **Rigidité des outils existants** : l'ajout de nouvelles règles ou l'évolution des référentiels nécessite souvent des développements spécifiques, limitant l'agilité des équipes et la réactivité face à l'évolution des normes.

Le langage Ada, par sa richesse syntaxique, son typage fort, ses mécanismes avancés (généricité, multitâche, héritage, visibilité fine), exacerbe ces difficultés. Les règles de codage dans Ada sont particulièrement variées : certaines sont locales (absence d'instruction `abort`), d'autres globales (absence d'appel indirect à une procédure critique, respect de l'initialisation des variables, etc.), et peuvent nécessiter de parcourir des graphes de dépendances complexes.

Face à ces constats, nous posons l'hypothèse qu'une approche fondée sur les bases de données graphe, capable de modéliser explicitement les relations sémantiques et structurelles du code, permettrait de lever ces verrous. L'objectif général de cette thèse est donc de concevoir, implémenter et évaluer une méthode de vérification automatique des règles de codage Ada s'appuyant sur une modélisation graphe, afin d'améliorer :

- la **performance** de l'analyse sur des bases de code de grande taille ;
- l'**expressivité** des règles, y compris pour des cas complexes ou globaux ;
- la **robustesse** et la reproductibilité de la vérification dans des contextes industriels variés.

Les objectifs spécifiques incluent : la définition d'un schéma de graphe adapté à Ada, la traduction des règles en requêtes de pattern matching (Cypher), l'évaluation expérimentale sur un corpus de 134 projets Ada open-source, et la comparaison quantitative avec les outils de l'état de l'art.

État de l'art

L'analyse statique du code source s'appuie historiquement sur plusieurs représentations intermédiaires : l'AST décrit la structure syntaxique du programme ; le Control Flow Graph (CFG) modélise les chemins d'exécution possibles ; le Program Dependency Graph (PDG) explicite les dépendances de données et de contrôle. Les graphes d'appels (Call Graph (CG)) permettent de visualiser les relations d'appel entre sous-programmes.

Plusieurs outils de vérification statique existent pour Ada :

- **AdaControl** : outil puissant, capable d'exprimer un grand nombre de règles, mais dont l'architecture repose sur une analyse locale de l'AST. Il montre ses limites sur les très grands projets et pour des règles nécessitant de traverser plusieurs unités de compilation.
- **GNATcheck** : intégré à la chaîne GNAT, il permet la vérification de règles standardisées, mais reste limité en termes d'expressivité et d'extensibilité.
- **Autres outils** : CodePeer (analyseur d'exécution symbolique), Polyspace (analyse abstraite), etc., qui ciblent des aspects complémentaires (détection de bugs, preuves formelles) mais ne répondent pas toujours aux besoins de vérification fine des règles de codage.

La littérature met en évidence plusieurs limites : difficulté à exprimer des règles globales, manque de scalabilité, rigidité des architectures. Des travaux récents ont proposé l'usage de graphes de propriétés (property graphs) pour l'analyse statique, notamment via le concept de Code Property Graph (CPG) : ces modèles englobent l'AST, le CFG et le CG dans un même graphe orienté, permettant des requêtes de pattern matching puissantes [50]. Ces approches ont montré leur efficacité en sécurité (détection de vulnérabilités, analyse de dépendances), mais restent peu explorées pour la vérification de règles de codage.

En synthèse, si les outils existants couvrent une large part des besoins industriels, ils peinent à répondre aux exigences de scalabilité, d'expressivité et d'agilité imposées

par l'évolution des bases de code et des référentiels de règles. Les avancées récentes en modélisation de graphe ouvrent de nouvelles perspectives pour dépasser ces verrous, mais nécessitent une adaptation fine aux spécificités du langage Ada et aux contraintes industrielles.

Méthodologie

L'approche proposée repose sur une architecture en plusieurs étapes visant à représenter le code Ada sous forme de graphe de propriétés exploitable par des requêtes de pattern matching. La méthodologie se décline selon les axes suivants :

Architecture de la solution

La chaîne de traitement se compose des étapes suivantes :

1. **Extraction de l'AST** : utilisation de librairies tels que Ada Semantic Interface Specification (ASIS) pour obtenir une représentation syntaxique complète du code source.

2. **Enrichissement sémantique** : ajout d'informations sur les types, les dépendances, les relations d'appel et d'héritage, issues de l'analyse statique et de la résolution des symboles.

3. **Génération du graphe** : transformation des entités et relations extraites en nœuds et arêtes d'un property graph, stocké dans une base Neo4j.

4. **Traduction des règles** : conversion des règles de codage en requêtes Cypher, permettant de rechercher des motifs précis dans le graphe.

5. **Analyse et restitution** : exécution des requêtes, collecte des résultats, génération de rapports détaillés.

Modélisation du graphe

Le schéma du graphe est conçu pour capturer la richesse syntaxique et sémantique d'Ada :

- **Types de nœuds** : procédures, fonctions, packages, types, variables, constantes, tâches
- **Types de relations** : appels (`CALL`), lien d'AST (`IS_ENCLOSED_IN`), le type correspondant (`IS_OF_TYPE`), etc.

Certaines relations complexes (ex : IS_ANCESTOR_OF, IS_PROGENITOR_OF) sont calculées directement dans la base, offrant une expressivité accrue.

Traduction des règles en requêtes Cypher

Les règles de codage sont exprimées sous forme de requêtes Cypher, permettant de détecter des motifs structurels ou sémantiques. Les requêtes Cypher combinent une clause `MATCH` décrivant le motif à rechercher, des prédicats `WHERE` optionnels pour filtrer les nœuds ou relations selon leurs labels et propriétés, et une clause `RETURN` spécifiant les éléments à retourner.

À titre d'illustration, considérons une procédure Ada `Cat_Laser_Pointer`. Nous pouvons utiliser Cypher pour récupérer tous les sous-programmes qu'elle appelle :

Code 0.1 — Requête Cypher illustrant les relations d'appel

```
MATCH (p:A_PROCEDURE_DECLARATION { name:
    "Cat_Laser_Pointer" })-[:CALLING]->(callee)
RETURN callee
```

Cette requête se décompose ainsi :

- `(p:A\_PROCEDURE\_DECLARATION \{ name: "Cat\_Laser\_Pointer" \})` désigne un nœud avec le label `A\_PROCEDURE\_DECLARATION` et une propriété `name` égale à `"Cat\_Laser\_Pointer"`.

- `-[:CALLING]->` représente une relation dirigée de type `CALLING`, indiquant que `p` appelle un autre sous-programme.

- `(callee)` introduit une variable liée à chaque sous-programme appelé par `p`.

- `RETURN callee` retourne tous les nœuds correspondant aux appelés.

Ces mêmes éléments de base (nœuds avec labels et propriétés, relations typées, et variables liées par pattern matching) permettent d'exprimer des règles de codage plus sophistiquées. Par exemple :

Code 0.2 — Absence d'instruction "abort"

```
MATCH (n:Statement {kind: 'Abort_Statement'}) RETURN n
```

Code 0.3 — Contrôle le nombre maximum de parents autorisé pour un type

```
// Parameters of the query
:params { "minNbParents": 2 }

// The query
MATCH (typeDecl)<-[r:IS_PROGENITOR_OF|IS_ANCESTOR_OF]-(parent)
WITH typeDecl, count(r) as nbParents
WHERE nbParents ⩾ $minNbParents
RETURN typeDecl, nbParents
ORDER BY typeDecl.filename, typeDecl.line, typeDecl.column
```

Corpus expérimental

L'approche a été évaluée sur un corpus de 134 projets Ada open-source, couvrant une grande diversité de tailles (de quelques milliers à plusieurs millions de lignes) et de complexités. Les critères de sélection incluent : version d'Ada maximum = Ada 2012, compilation sur notre architecture de test (Linux x86-64).

Pour garantir une évaluation complète, nous avons analysé l'utilisation des fonctionnalités du langage Ada dans le corpus. L'analyse quantitative révèle une couverture étendue : le polymorphisme paramétrique (59 projets déclarant des unités génériques, 992 instanciations publiques), la programmation orientée objet (69 projets utilisant des hiérarchies de types étiquetés avec des profondeurs d'héritage allant jusqu'à 7 niveaux), la gestion des exceptions (77 projets avec des instructions `raise`, totalisant 2889 occurrences), et l'utilisation intensive des types d'accès (8302 utilisations de l'attribut `'Access` dans 54 projets). Cependant, le corpus présente une couverture limitée des fonctionnalités de concurrence (seulement 3 projets avec des déclarations de tâches, 9 projets avec des objets protégés), reflétant la prédominance de la logique séquentielle dans l'écosystème Ada open-source disponible via Alire.

Critères d'évaluation

L'évaluation de la méthode repose sur les axes suivants :

- **Temps d'analyse** : mesure du temps total et par règle, comparaison avec AdaControl et GNATcheck.
- **Scalabilité** : capacité à traiter efficacement de très grands projets.
- **Couverture des règles** : proportion de règles vérifiables, expressivité des requêtes.
- **Robustesse** : capacité à gérer des cas limites, reproductibilité des résultats sur des codes hétérogènes.

Résultats principaux

Performances quantitatives

Le temps d'analyse moyen par projet, pour l'ensemble des règles, s'établit à environ 39 s 44 ms secondes pour notre approche (Cogralys), contre 4 min 5 s 827 ms secondes pour AdaControl et 11 min 36 s 86 ms secondes pour GNATcheck sur le même corpus. L'approche graphe démontre des accélérations significatives allant de 6.30 × à 17.83 × par rapport aux outils traditionnels, selon la complexité des règles et la taille du code.

Nous avons également identifié les facteurs clés influençant les performances de l'analyse statique à travers plusieurs études de cas :

- **Dominance du surcoût d'initialisation :** Pour de nombreux projets, notamment les plus grands comme AICWL (avec une profondeur d'héritage de 5 niveaux et plus de 100 instanciations génériques), le coût de construction de l'AST/graphe domine le temps de vérification des règles.
- **Complexité structurelle :** La complexité algorithmique et les relations inter-composants (comme dans APDF) augmentent le temps d'analyse indépendamment des fonctionnalités spécifiques du langage.
- **Complexité des données :** Les grandes définitions de données statiques (comme dans le projet Emojis avec ses tables d'initialisation massives) peuvent être aussi coûteuses que la logique complexe en raison du volume de nœuds AST générés.

Table 1: Comparaison des performances globales

Métrique	Cogralys	AdaControl	GNATcheck (1)	GNATcheck (32)
Temps d'analyse	39 s 44 ms	4 min 5 s 827 ms	11 min 36 s 86 ms	11 min 27 s 799 ms
Relatif au meilleur	**Référence**	6.30 ×	17.83 ×	17.62 ×
Initialisation	3 h 47 min 31 s 709 ms	7 min 41 s 801 ms	1 min 26 s 762 ms	1 min 31 s 454 ms
Exécution totale	3 h 48 min 10 s 753 ms	11 min 47 s 628 ms	13 min 2 s 848 ms	12 min 59 s 253 ms
Total relatif	19.35 ×	**Référence**	1.11 ×	1.10 ×

Scalabilité et performance par type de règle

L'analyse révèle deux tendances distinctes pour les projets de moins de 10 000 lignes de code[1], avec des comportements de performance différents selon la structure du projet. Pour les grands projets, l'avantage de l'approche graphe devient particulièrement significatif.

Les performances varient également selon le type de règle :

- **Règles locales** : accélérations jusqu'à 205 × par rapport à AdaControl
- **Règles intermédiaires** : accélérations d'environ 140 × par rapport à AdaControl
- **Règles globales** : améliorations significatives mais plus modérées

[1] Les résultats pour les projets de plus de 10 000 lignes de code sont présentés dans le chapitre 3.

Table 2: Analyse des performances par type de règle

Règle	Cogralys	AdaControl	GNATcheck (1)	GNATcheck (32)
Règles locales:				
Abort Statements	3 s 400 ms	10 min 900 ms	51 s 900 ms	52 s 300 ms
Blocks	2 s 100 ms	9 min 46 s 900 ms	52 s 200 ms	52 s 200 ms
Constructors	7 s 700 ms	9 min 13 s 10 ms	1 min 7 s 300 ms	1 min 7 s 200 ms
Enumeration Rep. Clauses	1 s 90 ms	9 min 36 s 100 ms	41 s 900 ms	42 s 500 ms
Renamings	1 s 100 ms	9 min 53 s 500 ms	52 s 90 ms	52 s 300 ms
Slices	1 s 700 ms	9 min 56 s 400 ms	16 min 29 s 200 ms	16 min 21 s 800 ms
Total Local	17 s 90 ms	58 min 26 s 810 ms	20 min 54 s 590 ms	20 min 48 s 300 ms
Moyenne d'accélération	**Référence**	205 ×	73 ×	73 ×
Règles intermédiaires:				
Abstract Type Declarations	4 s 80 ms	9 min 29 s 700 ms	51 s 900 ms	52 s 100 ms
Total Intermédiaire	4 s 80 ms	9 min 29 s 700 ms	51 s 900 ms	52 s 100 ms
Moyenne d'accélération	**Référence**	140 ×	13 ×	13 ×
Règles globales:				
Too Many Parents	2 s 300 ms	9 min 56 s 400 ms	58 s 800 ms	59 s 700 ms
Total Global	2 s 300 ms	9 min 56 s 400 ms	58 s 800 ms	59 s 700 ms
Moyenne d'accélération	**Référence**	259 ×	25 ×	26 ×

Analyse préliminaire de la précision

Bien qu'une analyse complète des vrais/faux positifs et négatifs n'ait pas été menée, une investigation préliminaire sur quelques projets a confirmé que notre approche basée sur les graphes offre théoriquement une précision équivalente aux outils traditionnels basés sur l'AST. En effet, notre graphe est dérivé directement du même AST ASIS sans perte d'information structurelle, et les variations observées dans les comptages de messages proviennent de détails d'implémentation corrigeables plutôt que de limitations inhérentes à la représentation en graphe. Par exemple, notre revue initiale a révélé :

- **Faux négatifs :** causés par des requêtes Cypher incomplètes ne prenant pas en compte toutes les constructions du langage.

- **Faux positifs :** résultant de certains patterns du langage mal interprétés par une requête.
- **Différences de périmètre :** différences dans l'ensemble des fichiers analysés par chaque outil.

Synthèse comparative

En synthèse, l'approche graphe offre :

- Un gain de performance significatif sur l'analyse, particulièrement pour les grands projets
- Une expressivité accrue pour la formulation de règles complexes
- Une meilleure compréhension sémantique du code par rapport aux approches textuelles
- Un accès indexé direct aux éléments spécifiques sans nécessiter de parcourir l'intégralité du code

Analyse, discussion et limites

L'approche par base de données graphe apporte plusieurs avancées conceptuelles et techniques majeures. Sur le plan conceptuel, elle permet d'unifier la représentation des relations syntaxiques et sémantiques et facilite la traversée de dépendances complexes. La flexibilité du schéma de graphe autorise l'ajout progressif de nouveaux types de nœuds et de relations, favorisant l'évolution du référentiel de règles sans refonte majeure de l'outillage.

Sur le plan technique, l'utilisation de requêtes Cypher offre une expressivité inégalée pour la formulation de règles, y compris celles impliquant des parcours transversaux, des corrélations multi-modules ou des contraintes de visibilité.

Parmi les limites identifiées :

- **Coût d'initialisation** : la génération du graphe représente un surcoût important (3,8 heures en moyenne), rendant l'approche plus adaptée aux grands projets ou aux analyses répétées où ce coût peut être amorti.
- **Dépendance entre règles et structure du graphe** : il existe une dépendance entre les règles de codage implémentées et les arêtes nécessaires dans le graphe, nécessitant potentiellement des enrichissements du schéma pour de nouvelles règles.
- **Limitations liées à ASIS-for-GNAT** : l'utilisation d'ASIS introduit des contraintes de performance et de maintenance, suggérant une transition future vers libadalang.

Malgré ces limites, l'approche ouvre des perspectives prometteuses pour l'analyse statique de code à grande échelle.

Conclusion et perspectives

Cette thèse propose une nouvelle génération d'outils de vérification statique, fondée sur la modélisation graphe, capable de répondre aux défis de performance, d'expressivité et de robustesse posés par les systèmes logiciels critiques modernes. L'approche a démontré sa capacité à traiter efficacement de très grands projets et à exprimer des règles complexes.

Notre implémentation prototype (Cogralys) est disponible en open-source pour faciliter la reproductibilité et les recherches futures : Cogralys.

Les perspectives ouvertes par ces travaux sont multiples :

- **Transition vers libadalang** : pour supporter les fonctionnalités modernes d'Ada et surmonter les limitations d'ASIS-for-GNAT.

- **Analyse complète de précision** : évaluation approfondie des vrais/faux positifs et négatifs dans les messages rapportés.
- **Mises à jour incrémentales** : développement de capacités de mise à jour incrémentale de la base de données graphe pour réduire les surcoûts dans les workflows de développement itératif.
- **Intégration aux environnements de développement** : pour fournir un retour en temps réel sur les violations des règles de codage.
- **Extension à d'autres langages** : application de l'approche à d'autres langages de programmation avec des systèmes de types complexes, tels que C++, Rust ou OCaml. Pour les langages fonctionnels comme OCaml, des fonctionnalités telles que le pattern matching et les fonctions de première classe créent des graphes de flux de contrôle et de données complexes que notre schéma pourrait modéliser. Pour Rust, les concepts de propriété (ownership) et de durées de vie (lifetimes) pourraient être explicitement modélisés comme propriétés ou relations dans le graphe.

À terme, cette approche pourrait contribuer à l'émergence de standards ouverts pour la vérification statique avancée, et à la démocratisation de pratiques de qualité logicielle dans les secteurs les plus exigeants.

Contents

List of Tables

List of Figures

List of elements

Introduction

In the modern world, software has become ubiquitous, permeating every aspect of our daily lives. From the smartphones we carry in our pockets to the complex systems that power our industries, software is an integral and indispensable part of our society. As our dependence on software grows, ensuring its quality, reliability, and security becomes increasingly critical.

In many domains, such as aerospace, defense, healthcare, and automotive industries, softwares play a vital role in controlling and monitoring systems that directly impact human lives and safety [41]. In these contexts, even a seemingly minor bug or defect in the code can have severe and far-reaching consequences. History provides numerous examples of software failures that have led to catastrophic outcomes, such as the Therac-25 radiation therapy machine accidents [31], the Ariane 5 rocket explosion [34], and the Boeing 737 MAX crashes [26].

Software bugs can manifest in various forms, each presenting unique challenges and risks. Common types of bugs include programming errors, omission errors, configuration errors, and concurrency errors [20, 53, 37, 41]. As software systems become more complex and interconnected, the potential for bugs and vulnerabilities increases dramatically. This is particularly true for autonomous systems, such as self-driving cars, drones, and robots, where software is responsible for making critical decisions in real-time [29]. In these cases, the consequences of software failures can be catastrophic, highlighting the need for rigorous quality assurance and testing practices.

Static code analysis, particularly the verification of coding rules, has emerged as a crucial technique in the software development lifecycle. By systematically examining the source code without executing it, coding rule checkers can be used to identify and mitigate potential issues before they manifest in the final product. A wide range of issues, from simple coding style violations to more complex structural problems, can be uncovered by these tools [12].

However, traditional tools for coding rule verification often struggle to keep pace with the increasing complexity and scale of modern software systems [21]. These tools face challenges in efficiently analyzing large codebases, leading to long analysis times and limited scalability [50, 38]. A significant portion of the analysis time is spent on accessing all the information related to a code construct, which may be scattered across multiple files and, consequently, multiple Abstract Syntax Tree (AST). This poses a major challenge for AST-based methods, hindering their ability to efficiently verify coding rules that require navigating and integrating information from different parts of the codebase [39].

To address these challenges, in this thesis we designed a proof of concept to analyze how graph databases and pattern matching queries may enhance the scalability and performance of coding rule verification. By representing code as a graph, where nodes denote code constructs and edges represent relationships between them, we can efficiently store and query the code structure and dependencies. This graph-based representation allows for more expressive and efficient verification of coding rules compared to traditional AST-based methods.

We focus specifically on the Ada programming language for our investigation. Ada is known for its complex code structures and is widely used in safety-critical systems where adherence to coding standards is of utmost importance. We aim to advance the state-of-the-art in coding rule verification techniques by developing a graph-based representation of Ada code and evaluating its effectiveness through extensive benchmarking, thereby contributing to its practical application in real-world software development scenarios.

The main contributions of our thesis are as follows:

- Our development of a novel graph-based representation of Ada code using the Neo4j graph database, which enables efficient traversal and pattern matching queries for coding rules verification.
- Our creation and open-source distribution of a comprehensive benchmark corpus consisting of more than a hundred Ada projects of various sizes, providing a valuable resource for the research community to evaluate static analysis tools.
- Our comprehensive evaluation of the proposed approach on various Ada codebases, demonstrating significant improvements in verification speed and scalability compared to traditional AST-based tools.
- Our insights into the potential of graph databases and pattern matching queries for enhancing coding rule verification, with implications for their application to other programming languages and software engineering tasks.

We also make our prototype implementation (Cogralys) available as open-source to facilitate reproducibility and further research: Cogralys. The repository provides the licensing information.

We have organized the remainder of this thesis as follows: In Chapter 1, we provide a review of related literature on coding rule verification and graph databases, including an overview of graph query languages and Cypher (Section 1.3.4). In Chapter 2, we describe the methodology, detailing the selected coding rules (Section 2.2) and the relationships created in the graph (Section 2.1), and we illustrate Cypher-based rule expression with concrete examples (Chapter 2, Section 2.5). In Chapter 3, we present benchmark results and analyses. In Chapter 4, we discuss the implications of our findings and identify areas for future research. Finally, in Chapter 4.4, we synthesize the key contributions and reflect on the significance of our research outcomes.

Chapter 1

Definitions & Related Work

In this chapter, we provide an overview of the key background topics and related work relevant to our research. The chapter is divided into four main sections.

Section 1.1 defines fundamental concepts from graph theory that are essential for understanding graph-based code representations and analyses.

Section 1.2 introduces core concepts of static code analysis, with particular emphasis on coding rule verification and different code representation techniques.

Section 1.3 examines graph databases and their advantages for modeling interconnected data such as code structures, highlighting their applications in software engineering and coding rule verification.

Section 1.3.6 outlines the selection process for an appropriate Graph DataBase Management System (GDBMS) to efficiently store and query our graph-based framework.

1.1 Definition related to graph theory

In graph theory, a **graph** is an ordered pair $G = (V, E)$, where V is a finite set of vertices and E is a finite set of edges. Each edge connects exactly two vertices, forming the fundamental structure of the graph. Depending on whether edges possess directional attributes or not, a graph may be classified as either a directed or undirected graph.

Vertices represent distinct entities, while edges denote relationships between these entities. In directed graphs, edges have inherent directionality, designating one vertex as the start and another as the end. Graphs have diverse applications, from algorithm design in computer science to social network analysis, providing a rigorous framework for exploring complex relationships and interconnected systems.

An important concept inherent to graph theory is labeling, which involves assigning categorical identifiers or tags, known as **labels**, to vertices and edges. Vertex labeling associates each node with labels that capture shared characteristics among vertices, enhancing the graph's semantic structure and facilitating targeted retrieval of specific groups of vertices. Similarly, edge labeling assigns categorical tags to relationships between vertices, classifying distinct types of connections and contributing to the graph's contextual understanding.

A **Property Graph** is a specialized form of graph that incorporates properties for both vertices and edges. Specifically, it is a directed graph $G = (V, E)$ where each vertex $v \in V$ is associated with a collection of key-value pairs, referred to as node properties, encapsulating additional descriptive information about the entity. Likewise, each edge $e \in E$ contains edge properties, constituting analogous key-value pairs that provide further information about the relationship between connected vertices. This augmentation of node and edge properties enriches the graph with nuanced information beyond mere connectivity. The concept of labeling is often integral to Property Graphs, with vertices assigned labels based on common characteristics, enhancing the organization and retrieval of graph elements and facilitating efficient query capabilities within the graph database framework.

As an example, Figure 1.1 depicts a property graph used to represent a code database, where nodes represent code construct like `An_Expression` or `A_Defining_Name` and edges represent relationships like `Is Enclosed In`. These graph elements are labeled (`An_Expression`, `A_Defining_Name`) and attached with properties such as `filename` providing a rich and contextually nuanced depiction of the underlying domain.

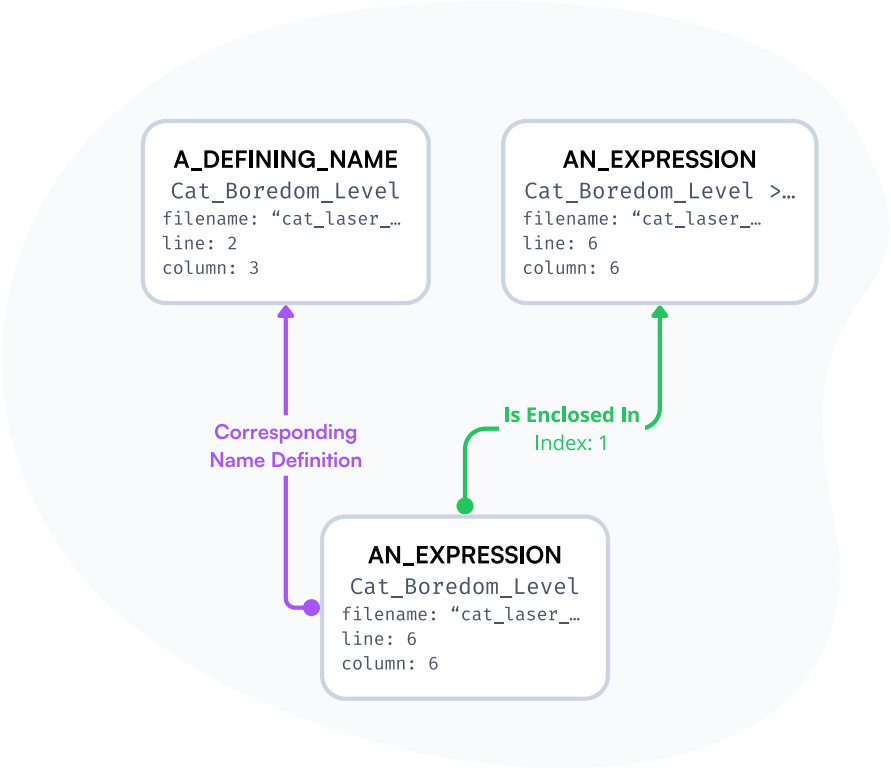


Figure 1.1: Example of a Property Graph

1.2 Static Code Analysis

1.2.1 Key Concepts in Static Code Analysis

To illustrate the concepts described in this section, let us consider the following Ada code snippet[1]:

[1] This example is an adaptation of the example provided by Yamaguchi in [50]

Code 1.1 — Example of Ada program to illustrate Static Code Analysis (SCA) concepts

```
1  procedure Cat_Laser_Pointer is
2     Cat_Boredom_Level : Natural;
3     Laser_Speed       : Natural;
4  begin
5     Cat_Boredom_Level := Observe_Cat;
6     if Cat_Boredom_Level > 8 then
7        Laser_Speed := Cat_Boredom_Level / 2;
8        Activate_Laser_Pointer (Laser_Speed);
9     end if;
10 end Cat_Laser_Pointer;
```

This code defines a procedure `Cat_Laser_Pointer` that aims to entertain a cat using a laser pointer. The procedure starts by observing the cat's boredom level, stored in the variable `Cat_Boredom_Level`. If the boredom level exceeds a threshold of 8, the laser pointer is activated with a speed proportional to the boredom level.

Abstract Syntax Tree (AST)

AST are typically one of the initial intermediate representations generated by code parsers in compilers, serving as a foundational element for creating various other code representations. These trees accurately depict the hierarchical structure of statements and expressions within programs.

ASTs are structured as ordered trees, where internal nodes symbolize operators (e.g., additions or assignments), and leaf nodes represent operands (e.g., constants or identifiers). From a graph theory perspective, an AST can be viewed as an acyclic graph, where edges exclusively represent nesting relationships between nodes. This makes ASTs a fundamental but simplified form of graph-based code representation. For example, Figure 1.2 illustrates a simplified AST from the example code presented earlier in Code 1.1[1].

While ASTs are effective for straightforward code transformations and have been used to detect semantically similar code segments, they fall short in more complex code

[1] The full AST can be found in appendix A.1

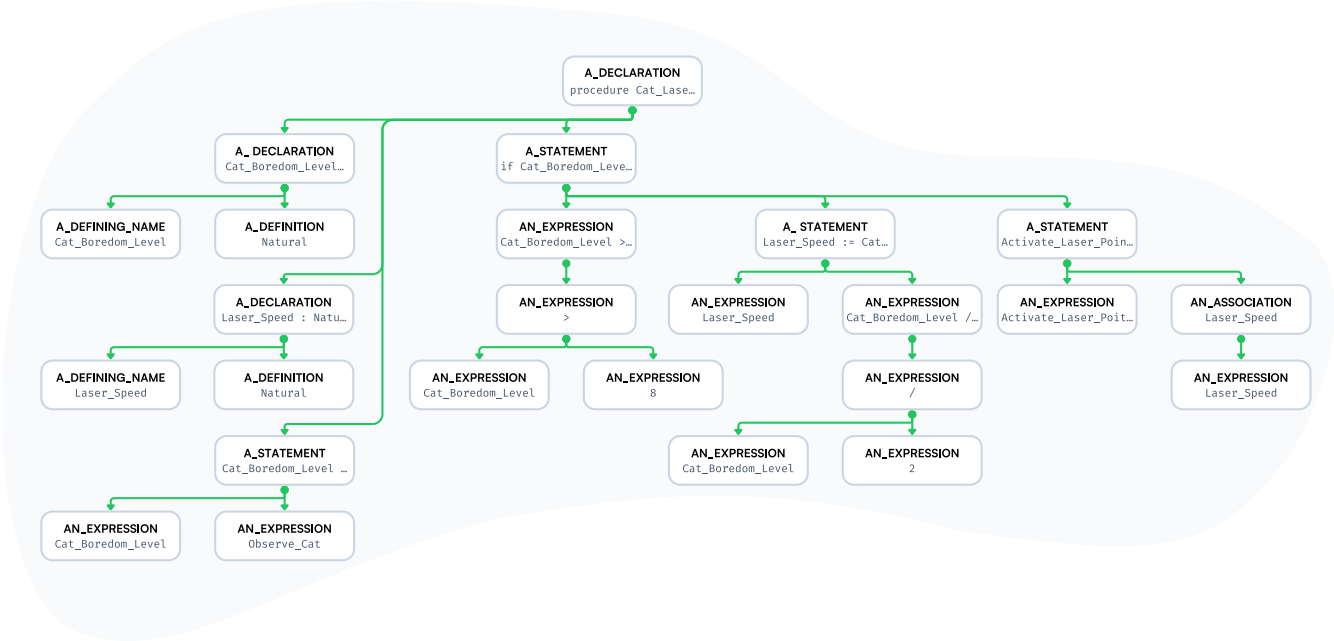


Figure 1.2: Simplified AST of ”Cat Laser Pointer” program

analysis tasks like identifying dead code or uninitialized variables. This limitation stems from their inability to explicitly represent control flow or data dependencies.

Consequently, static analysis tools often complement ASTs with additional graph-based structures such as Call Graph (CG), Control Flow Graph (CFG), and Program Dependency Graph (PDG). These supplementary representations, which will be discussed in subsequent sections, capture different aspects of program behavior and relationships that cannot be expressed through the simple nesting structure of an AST alone.

Call Graph (CG)

A Call Graph (CG) is a directed graph that represents the calling relationships between subprograms (functions or procedures) in a computer program. Each node in the graph represents a procedure, and each edge $f \longrightarrow g$ indicates that procedure f calls procedure g. In object-oriented programming, the nodes may represent methods, and edges may indicate not just direct calls but other references as well, such as virtual method invocations.

CGs can be static or dynamic. A static CG is built by analyzing the source code without executing the program. This type of analysis can be performed by examining the code and identifying all potential calls that could be made based on the program's structure. However, static analysis may identify calls that never actually occur during execution and may miss calls made through function pointers or virtual method invocations. Specifically, it fails to capture calls to subprograms that are dynamically determined at runtime, such as those made via function pointers, because these are not explicitly visible in the static code structure.

On the other hand, a dynamic CG is constructed by tracing the actual execution of the program. This approach provides a more accurate representation of the calls that occur during a specific run of the program, but it may miss calls that are not made during that particular execution.

Figure 1.3 displays the CG for the code example provided in 1.1.

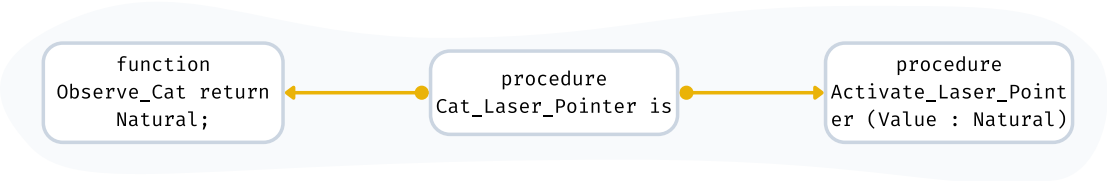


Figure 1.3: CG of "Cat Laser Pointer" program

Call graphs have various applications in software engineering and program analysis. They can be used for program understanding, as they provide a high-level view of the program's structure and the interactions between its components. Call graphs are also useful for debugging, optimization, and testing. For example, they can help identify unreachable code, recursive calls, or potential performance bottlenecks.

Control Flow Graph (CFG)

A Control Flow Graph (CFG) concretely outlines the sequence in which code statements are executed, along with the conditions necessary for a specific execution path to occur. In this representation, statements and predicates are depicted as nodes, linked by directed edges that signify control transfer [7]. Unlike in abstract syntax trees, these edges are not necessarily ordered, but each must be assigned a label of `true`, `false`, or ε. Typically, a statement node will have a single outgoing edge marked as ε, while a predicate node will feature two outgoing edges, each labeled to reflect the `true` or `false` outcome of the predicate. Constructing a CFG from an abstract syntax tree involves a two-step process: initially, structured control statements such as `if`, `while`, and `for` are used to create a preliminary CFG; subsequently, this graph is refined by incorporating unstructured control statements like `goto` or `break`. Figure 1.4 displays the CFG for the code example provided in 1.1.

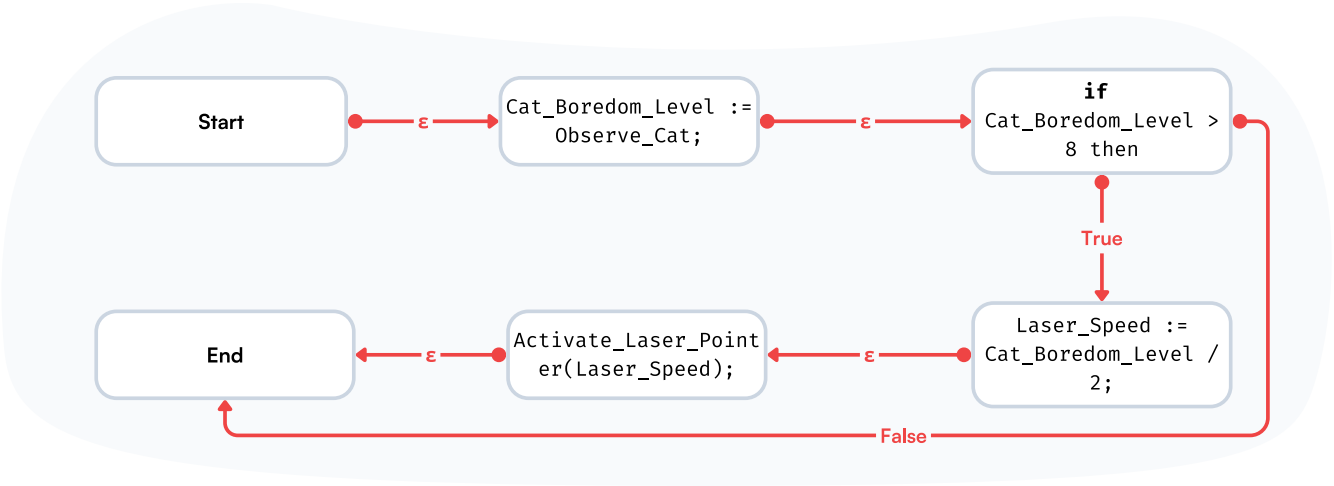


Figure 1.4: CFG of ”Cat Laser Pointer” program

CFGs are employed across various applications, particularly in security, such as detecting variants of known malicious software [18] and enhancing the efficacy of fuzz testing tools [46, 28]. They have also become essential in reverse engineering for facilitating program comprehension. However, while CFGs reveal the control flow within an application, they do not provide insights into data flow. Specifically, in the context of vulnerability analysis, this limitation means that CFGs are not straightforward tools for identifying statements that process data influenced by an attacker.

Difference between CG and CFG: CG and CFG are both composed of nodes and edges but serve different purposes within program analysis. The CG is interprocedural; it comprises nodes that represent subprograms (such as methods or functions), and edges that depict the caller-called relationships between these subprograms (for example, $A \rightarrow B$ indicates that subprogram A calls subprogram B). On the other hand, the CFG is intraprocedural, with nodes representing individual program statements, including subprogram calls and conditionals, while the edges trace the program's execution flow.

Program Dependency Graph (PDG)

Program Dependency Graphs (PDGs), initially introduced by Ferrante [17], were developed primarily to facilitate program slicing [22]. Program slicing is a technique for extracting a subset of a program's statements (a "slice") that is relevant to a particular computation or behavior of interest. The goal is to isolate the parts of the program that potentially affect the values computed at some point of interest, referred to as a slicing criterion. This process involves identifying all the statements and predicates in a program that influence the value of a variable at a specific statement. PDGs explicitly map out the dependencies between statements and predicates within a program. Specifically, the graph incorporates two types of edges: data dependency edges, which illustrate the impact one variable has on another, and control dependency edges, which represent how predicates affect variable values.

The construction of a PDG involves extracting these dependencies from a CFG. This is achieved by first identifying the variables defined and used by each statement, and then calculating the reaching definitions for each statement and predicate—an established challenge in compiler design.

For illustration, Figure 1.5 displays the PDG for the example code shown in 1.1. It is important to note that control dependency edges in a PDG do not simply mirror the control flow edges. Specifically, the PDG does not preserve the execution order of statements but clearly delineates the dependencies among statements and predicates.

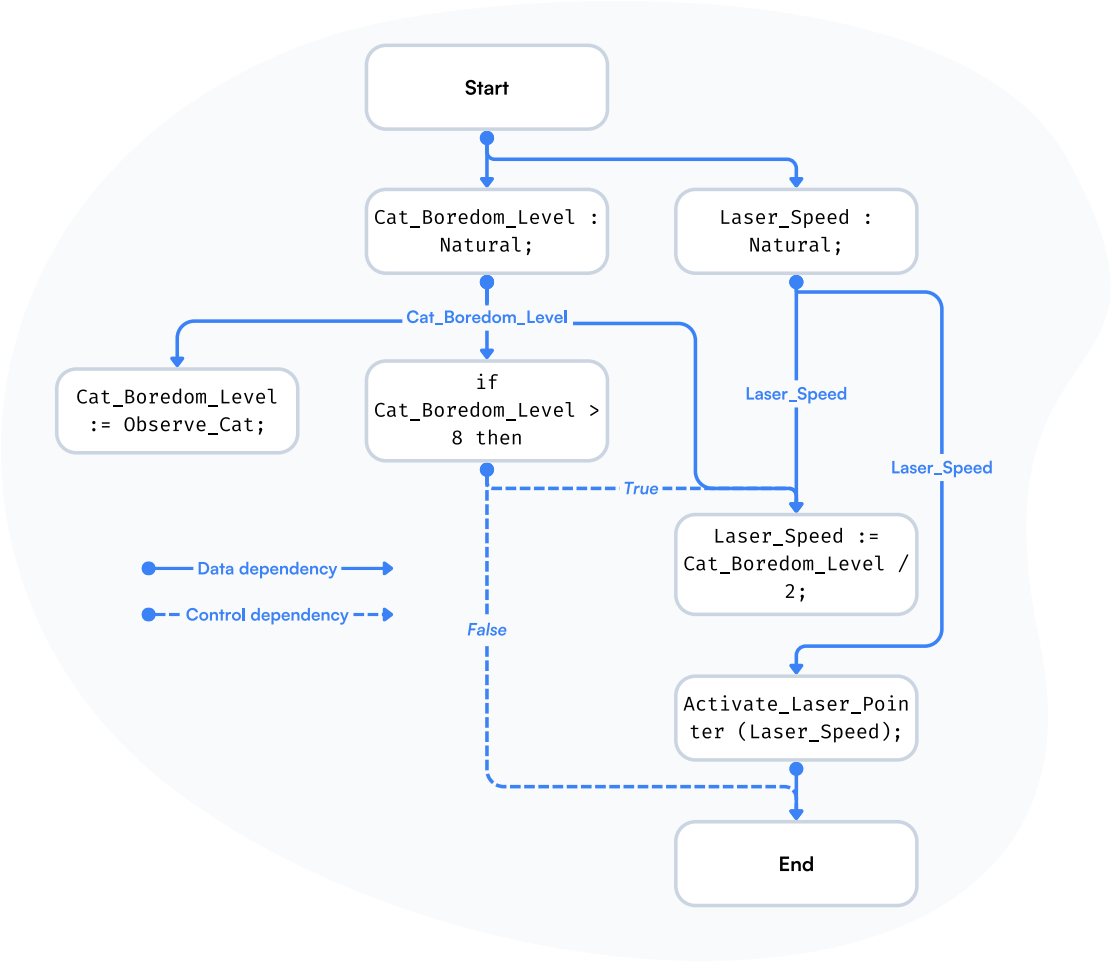


Figure 1.5: PDG of ”Cat Laser Pointer” program

Unified Graph-Based Code Representations

Code Property Graph (CPG): The Code Property Graph (CPG), introduced by Yamaguchi et al. [50], represents a significant advancement in code representation by combining multiple code perspectives into a single unified structure. It integrates:

1. The hierarchical syntactic structure from ASTs
2. The execution path information from CFGs
3. The data dependency relationships from PDGs

This integration creates a directed, attributed, and labeled multigraph where:

- Vertices correspond to the program's AST nodes
- Edges represent various relationships (syntactic, control flow, and data dependencies)
- Both nodes and edges can have properties (key-value pairs) for additional information

By unifying these perspectives, the CPG enables complex analyses that would be difficult with separate representations, simplifying the development of tools for code pattern detection, vulnerability identification, and program slicing.

Figure 1.6 shows the CPG for our example code from Code 1.1.

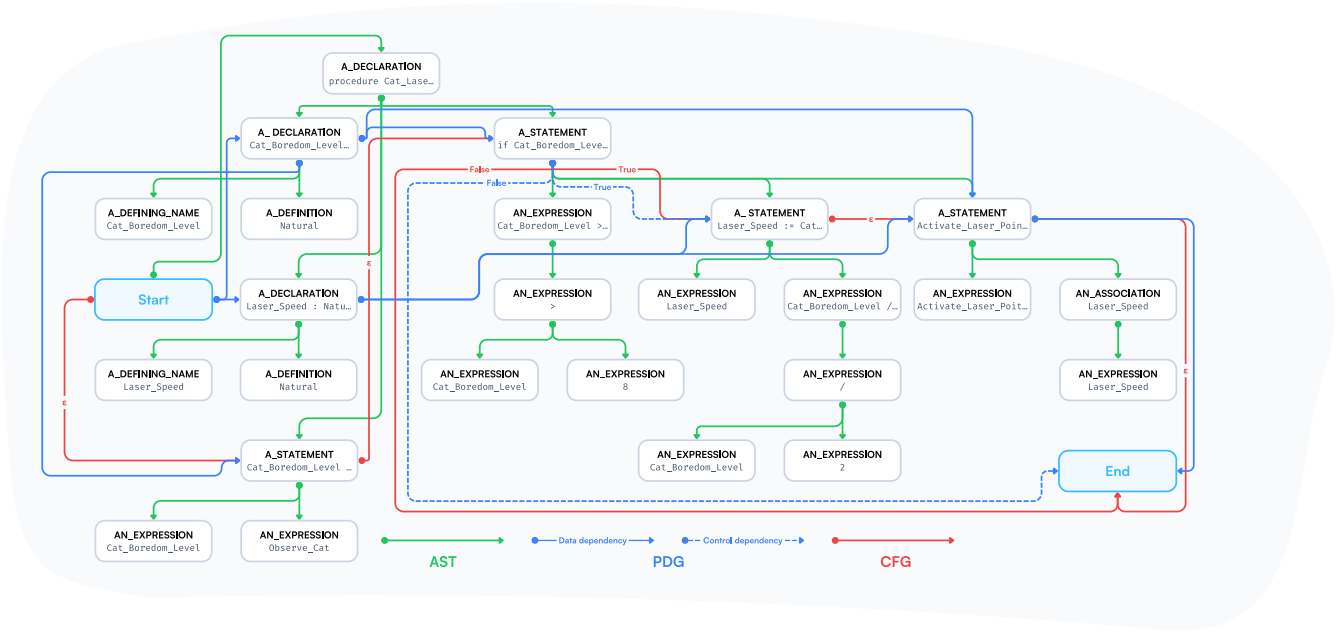


Figure 1.6: CPG of "Cat Laser Pointer" program

Extended Code Property Graph (ECPG): For our research, we extend the standard CPG by incorporating:

- Call graphs (CGs) to represent interprocedural relationships

- Custom relationships tailored to the needs of Ada coding rule verification

These extensions enhance the graph's ability to represent Ada-specific constructs and relationships, making it more suitable for our coding rule verification tasks. Figure 1.7 illustrates this extended representation for our example code (a complete version appears in Appendix A.2).

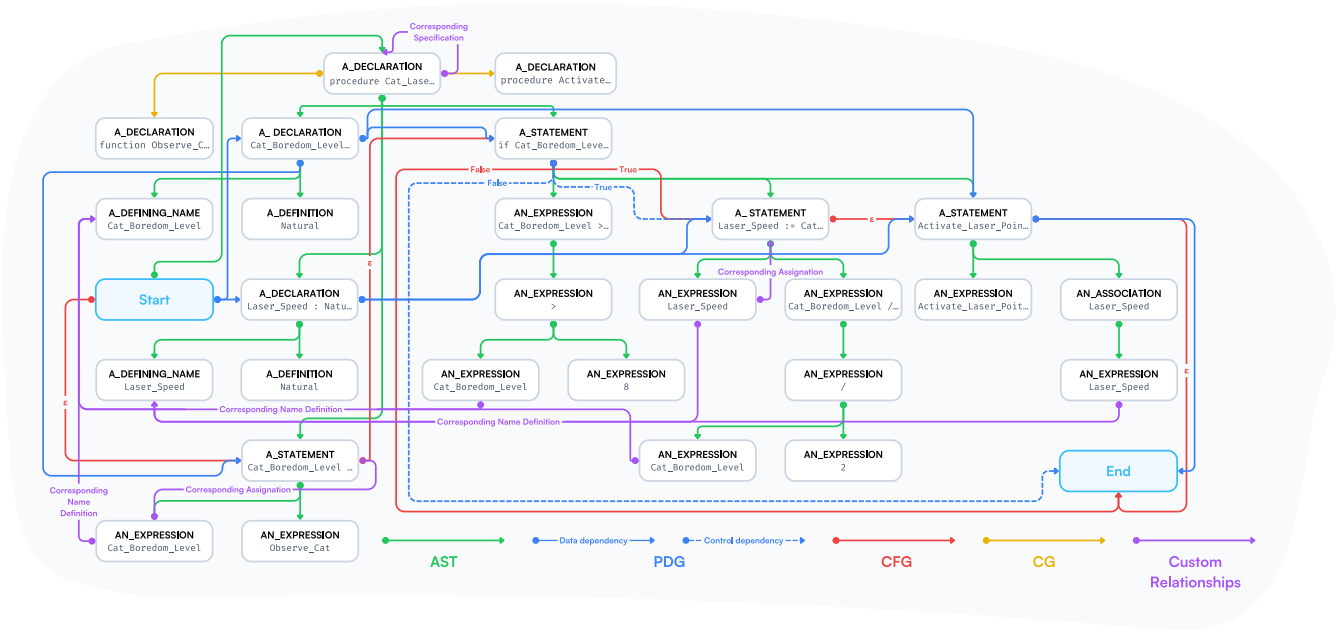


Figure 1.7: Extended CPG of "Cat Laser Pointer" program

Semantic Code Graph (SCG): The Semantic Code Graph (SCG) [11] represents a different approach to code representation, focusing on developer-relevant semantics rather than execution mechanics. Unlike the CPG which builds upon compiler-oriented representations, the SCG:

- Models concrete code entities that developers interact with (classes, methods, variables) - Explicitly represents semantic relationships (calls, declarations, inheritance, etc.) - Maintains precise source location linkage for each element - Supports extensibility for language-specific constructs

Figure 1.8 shows an SCG for our example code, highlighting the semantic relationships between code elements.

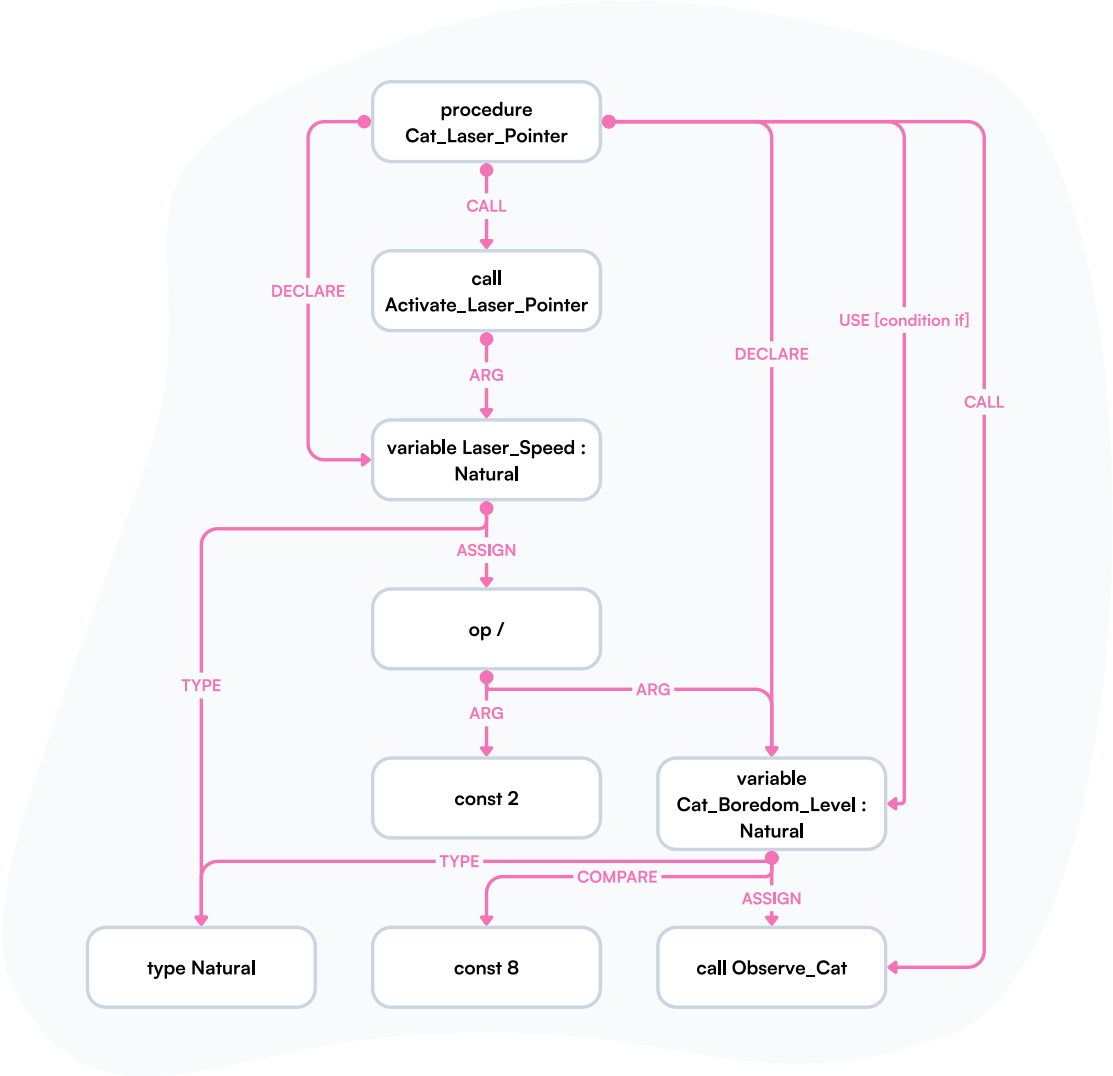


Figure 1.8: SCG of the ”Cat Laser Pointer” program

While both CPG and SCG aim to represent code as graphs, they differ in their focus: CPG prioritizes execution semantics for security analysis, while SCG emphasizes developer-oriented semantics for code comprehension.

The SCG offers several advantages over previous models. Unlike the PDG, which focuses on statement-level data and control dependencies, or the CPG, which integrates multiple lower-level representations, the SCG is centered on source-level entities and semantic relationships that are most relevant to software comprehension, refactoring, and quality assessment. Its extensibility and rich source mapping make it particularly

well suited for interactive tools, structural analyses, and actionable insight generation.

In summary, the SCG provides a comprehensive, source-centric, and semantically detailed representation of code structure and dependencies. This unified model serves as a foundation for advanced code analysis, visualization, refactoring, and software mining tasks, supporting both researchers and practitioners in understanding and maintaining complex software systems.

1.2.2 The Evolution of Static Code Analysis Tools

Static code analysis has a rich and varied history that traces back to the early days of software development. It involves examining the source code of a program without executing it, to identify potential defects and adherence to coding standards. Over the years, static code analysis has evolved significantly, driven by advancements in both theoretical foundations and practical implementations.

One of the pioneering tools in this field was Lint, developed in 1979 at Bell Labs for the C programming language [25]. Lint performed basic checks such as detecting unused variables, type mismatches, and potential memory leaks. This groundbreaking tool laid the foundation for the more sophisticated analysis techniques that would follow.

The 1990s and early 2000s saw significant advancements in static code analysis with the introduction of tools that could handle increasingly complex programming paradigms. Tools like LCLint, later renamed Splint, were developed to detect potential errors in C programs, focusing on improving security and supporting new programming paradigms [16, 30].

This period also saw the emergence of abstract interpretation, a mathematically rigorous approach that reasons about program behavior by abstracting away details while preserving essential properties [13]. Tools such as Polyspace and Astrée utilized this technique to detect runtime errors and prove the absence of certain classes of bugs in safety-critical software [10, 14].

Since the early 2010s, static code analysis has continued to evolve with the integration of these tools into Continuous Integration / Continuous Development (CI/CD) pipelines, becoming a standard practice. This integration enables real-time feedback and early detection of issues during the development process [33, 32].

The use of graph-based representations, particularly the CPG, represents a notable

advancement in static code analysis. Introduced as a novel code representation, CPG combines various aspects of code structure and semantics into a single, unified graph, allowing for more efficient and scalable analysis, especially for large codebases [35].

The field has also seen the integration of machine learning (ML) and artificial intelligence (AI) to enhance static analysis. Machine learning models, such as those fine-tuned with CodeBERT, are being used to improve the generality and accuracy of defect detection. These models can learn from historical data to identify patterns and predict potential issues, thereby reducing the reliance on manually crafted rules and heuristics [27].

From its origins characterized by elementary syntactic checks to the current state featuring advanced, AI-enhanced tools, coding rule verification has established itself as an essential element of modern software development practices. Traditionally, the principal architecture underpinning coding rule verification has been based on AST. This is particularly evident in the Ada static analyzers that we compare in this study (AdaControl and GNATcheck). While research has demonstrated the benefits of graph-based analysis for languages like Java [40] (using CPG) and C++ [39] (using custom representation), our research represents the first application of graph-based analysis for coding rule verification of Ada programs.

1.2.3 Ada programming language as a case of study

We selected the Ada programming language as the target for this research due to its inherent complexity and the sophistication of its applications. This choice is also driven by the needs of the company that funded this thesis, Adalog, which specializes in Ada development and quality assurance tools.

Origins and Evolution of Ada

Ada's development began in 1974 when the US Department of Defense identified significant inefficiencies from using different programming languages for each application and maintaining programs written in obsolete languages. This led to specifications for a single, flexible language capable of handling diverse applications from missile guidance to management systems.

The language design contract was awarded in 1977 to a team from CII-Honeywell Bull led by Jean Ichbiah, who drew inspiration from their previous work on LIS (Langage

d'Implémentation de Systèmes). The result was Ada 83, standardized in the United States by ANSI (ANSI/MIL-STD-1815A-1983) and later internationally as ISO 8652:1987.

Subsequent revisions have kept Ada at the forefront of programming language development:

- Ada 95 (ISO/IEC 8652:1995) - The first internationally standardized object-oriented language
- Ada 2005 - Added contract mechanisms and enhanced container libraries
- Ada 2012 - Introduced comprehensive contract-based programming
- Ada 2022 - Further refinements and modern programming features

The GNAT compiler, based on GNU GCC technology, was developed to fully support the Ada standards, including all optional annexes.

Key Features and Capabilities

Ada represents a synthesis of the best elements from classical imperative and procedural languages, designed specifically to reduce software costs across the entire development lifecycle. Though contemporary with C++, Ada offers distinctive modern features:

- Strong static typing system with comprehensive compile-time checks
- Modular design through packages with fine-grained visibility control
- Clear, Pascal-inspired syntax emphasizing readability and maintainability
- Sophisticated object-oriented programming with interfaces similar to Java
- Advanced genericity beyond templates in other languages
- Integrated multitasking and multi-core support
- Real-time capabilities with tasks, protected objects, and interrupt handling
- Comprehensive standard libraries with strong typing guarantees
- Multi-language interoperability for system integration

- The Ravenscar profile for safety-critical real-time systems
- Contract-based programming for formal verification

Complexity of the Ada Language

Despite its classical appearance, the Ada language is extremely difficult to compile. The language's Backus-Naur Form (BNF), provided in the standard, does not allow for the direct generation of a syntax tree, notably due to the need to resolve identifiers to remove ambiguities between certain constructions. Moreover, a simple AST, even fully resolved, is insufficient for the needs of analysis tools. Ada Semantic Interface Specification (ASIS) [1] (whose name expresses well that it is a semantic Application Programming Interface (API)) provides access to the fully decorated AST.

For example, expressions like `V := A (B);` appear syntactically identical at first glance, but could represent many different semantic constructs:

- `A` is a function, `B` is a parameter
- `A` is an array, `B` is an index
- `A` is a type, `B` is converted to `A`
- `A` is a function without parameters returning an array, `B` is an index
- `A` is a pointer to a function, `B` is a parameter
- And many more combinations of previous examples using pointers...

While these cases would produce similar initial syntax trees during parsing, distinguishing between them requires additional semantic analysis phases beyond simple AST construction. The compiler or parsing framework must perform name resolution, type checking, and overload resolution to determine the actual meaning of the expression. This complexity in Ada's semantics means that a simple AST is insufficient for proper code analysis.

It is, therefore, necessary to resolve overloads, establish links between elements (such as between an identifier and its declaration), and construct lists of elements that meet certain criteria (such as the set of different pragmas or aspects applicable to a declaration). All these elements are, of course, subject to visibility rules, overloading, etc.

While this complex parsing and semantic analysis is handled by compilers or parsing frameworks like ASIS or libadalang [3], the similarity of AST[1] structures for semantically different constructs makes static analysis more challenging, as tools must carefully consider these semantic differences to avoid false positives in their analysis results. To address this challenge, our approach leverages a CPG.

The choice of Ada as our target language for this study is motivated by the fact that the new architecture studied is likely to bring a more significant gain with a language whose compilation is complex. Thus, choosing Ada first will allow us to realize the relevance of the choices made and the effectiveness of the project's architecture.

Furthermore, the operator overloading system is also a mechanism present in other languages, such as Python. Supporting Ada will therefore simplify the work for other languages.

Industrial Applications and Context

Ada has achieved significant success in diverse fields requiring high reliability and safety:

- **Transportation Systems**: The Paris Metro Line 14, in operation for over 20 years, uses Ada for its control systems. This success led to Ada's adoption for the New York City Subway and Singapore's Circle Line.
- **Aerospace**: Ada powered the Ariane 4 and 5 programs, with its contract-based programming capabilities leading to adoption for Ariane 6.
- **Air Traffic Control**: EuroControl's flight plan management system (handling up to 34,000 flights daily) comprises over 2.3 million lines of Ada code, where new features are added every day, requiring daily code analysis to ensure reliability.
- **Other Domains**: Ada is also used in management systems, CAD applications, medical devices, and linguistic processing, with many applications operating for more than a decade.

In the context of static analysis tools, AdaControl (developed by Adalog) represents an important coding rule verification tool for Ada. It relies on ASIS to analyze source

[1] It should be emphasized that identical code expressed in various programming languages does not yield equivalent ASTs, owing to differences in language constructs.

code. While ASIS provides standardized access to semantic information, the main implementation (ASIS-4-GNAT by AdaCore) faces significant challenges:

- Performance issues due to "TreeSwapping", where analyzing a compilation unit requires loading the semantic tree of all its dependencies even when unnecessary for the coding rules being verified
- Tight coupling with specific GNAT compiler versions (e.g., ASIS 2019 works only with GNAT 2019), creating maintenance and compatibility issues

These limitations restrict both performance and adaptability, highlighting the need for a more flexible approach to Ada static analysis that maintains high quality while overcoming these constraints.

Conclusion

The complexity of Ada, combined with the limitations of current analysis tools, presents several significant challenges that this research aims to address:

- Developing a more efficient and flexible analysis approach that isn't constrained by compiler-specific dependencies
- Creating a solution that can handle Ada's complex semantic constructs while maintaining good performance
- Building an architecture that can effectively manage the rich type system and overloading mechanisms present in Ada

While these challenges are identified in the context of Ada, they are not unique to this language. Many modern programming languages share similar features. Therefore, the solutions developed in this research could potentially be adapted to address similar challenges in other programming languages, particularly those with sophisticated type systems and complex semantic rules.

1.3 Graph Databases for Code Analysis

Code analysis tools have traditionally maintained code representations in memory using custom data structures. However, as codebases grow in size and complexity, database-backed approaches have emerged as a scalable alternative. This section examines the application of graph databases to code analysis, their comparative advantages, and their query capabilities.

1.3.1 Database-Backed Code Analysis Approaches

Several established tools demonstrate the viability of database-backed code analysis:

- **Relational database approaches**: CodeQL [19] (formerly SemmleQL) uses relational structures for vulnerability detection, SciTools' Understand [44] employs SQLite for code intelligence, and SonarQube [45] utilizes relational databases for quality management.
- **Graph database approaches**: Joern [51], created by the inventor of CPG, pioneered graph-based vulnerability detection using a custom graph database [52].

Our approach adapts graph databases specifically for coding rule verification rather than vulnerability detection. By selecting an established GDBMS, we leverage mature graph operations and declarative query languages, allowing us to express coding rules as patterns rather than procedural algorithms.

1.3.2 Graph Database Fundamentals

Unlike relational databases with their table-based model or document databases with their hierarchical collections, graph databases store and manage data as interconnected nodes and relationships. This structure is specifically optimized for traversing complex relationships.

The core elements of a graph database are:

- **Nodes**: Store the primary data entities with labels indicating their types or roles

- **Relationships**: Connect nodes with directional, typed edges
- **Properties**: Key-value pairs that can be attached to both nodes and relationships [8]

This model naturally represents interconnected data like code structures, enabling efficient pattern-based queries without the complex join operations required in relational databases. Modern graph databases maintain specialized indexes on nodes, relationships, and properties to optimize both storage and query performance.

1.3.3 Advantages for Code Analysis Applications

Graph databases provide particular benefits for code analysis when compared to traditional relational databases:

- **Natural representation of code relationships**: Graph databases directly model the inherently connected nature of code elements, representing relationships like ”calls,” ”inherits from,” or ”depends on” as first-class entities. This avoids the complex join tables required in relational models and aligns with how developers conceptualize code.
- **Path-based query efficiency**: Many coding rules require tracing sequences of relationships (e.g., finding all possible execution paths to a critical function). Graph databases excel at traversal operations that would require multiple recursive joins or stored procedures in relational databases.

Vendors of graph database management systems, such as Neo4j, explicitly aim to provide optimized implementations of standard graph algorithms, particularly for complex operations such as graph partitioning and path-finding. While these claims are often supported by technical documentation and blog articles [23], it is important to note that such sources are not peer-reviewed scientific publications, and their reported performance results should not be interpreted as independently validated evidence. Nonetheless, these efforts illustrate the ongoing focus within the graph database community on efficient algorithmic support for advanced graph analytics tasks, which is directly relevant to applications in code analysis.

- **Flexible schema evolution**: As coding standards evolve or new language features emerge, graph databases allow incremental model enhancement without

disruptive schema migrations [48]. This is particularly valuable for long-term code analysis projects covering multiple language versions.

- **Pattern matching capabilities**: Graph databases provide built-in pattern matching that aligns naturally with the structural patterns that coding rules aim to identify or enforce.

These advantages are especially relevant for complex coding rules that require correlating information across different parts of the codebase, such as detecting resource leaks, validating architectural constraints, or ensuring proper exception handling.

1.3.4 Graph Query Languages for Code Analysis

Effective code analysis requires powerful query capabilities to express complex patterns and relationships. Graph query languages provide specialized syntax for traversing and pattern matching in code representations.

Major Graph Query Languages

Two dominant query languages have emerged in the graph database ecosystem:

- **Cypher** (Neo4j): A declarative pattern matching language where queries express what patterns to find rather than how to find them. Its ASCII-art syntax for describing nodes and relationships (e.g., `(a)-[:CALLS]->(b)`) makes queries visually intuitive and readable, particularly for expressing structural code patterns.
- **Gremlin** (Apache TinkerPop): An imperative traversal language that uses method chaining to specify graph navigation steps. Its procedural approach offers fine-grained control over traversal operations and is often preferred for complex algorithmic analyses.

Standardization efforts are underway with the Graph Query Language (Graph Query Language (GQL)) [9, 15], which is being developed as an ISO standard (ISO/IEC 39075) to provide a unified approach to graph querying.

Applications in Software Engineering

Graph query languages enable developers and analysts to express complex code analysis tasks that would be difficult with traditional query languages [39], including:

- Architecture compliance verification
- Dependency analysis and impact assessment
- Security vulnerability detection
- Code quality rule enforcement
- Identification of design patterns and anti-patterns

For our coding rule verification, Cypher's declarative, pattern-based approach aligns naturally with how coding rules are conceptualized—as structural patterns to identify or enforce. This makes it particularly well-suited for expressing coding standards in a way that is both efficient to execute and comprehensible to developers.

1.3.5 Evolution of Graph-Based Code Analysis Applications

The application of graph databases to code analysis has evolved significantly over the past decade, with each major contribution addressing different aspects of software engineering challenges:

- **Pioneering Work (2013)**: Urma and Mycroft [47] were the first to demonstrate the practical scalability of graph databases for large codebases. Using Neo4j, they successfully analyzed multi-million-line programs while maintaining detailed source information, primarily focusing on program structure and evolution tracking.
- **Security Applications (2014-2016)**: Following the introduction of the CPG by Yamaguchi et al. [50], graph-based approaches were primarily applied to security vulnerabilities detection in C/C++ code. Their work demonstrated how graph traversal could efficiently identify subtle security patterns that would be difficult to detect with traditional approaches.

- **Comparative Studies (2018)**: Ramler et al. [39] conducted comprehensive evaluations of graph database technologies for source code representation. Their work critically assessed various use cases including dependency analysis and architecture recovery, providing evidence-based insights into both strengths and limitations.

- **Language-Specific Solutions (2020-2023)**: More recent applications have targeted specific programming languages with tailored approaches. Rodriguez-Prieto et al. [40] developed specialized graph structures for Java analysis, while Borowski et al. [11] created the SCG model optimized for object-oriented languages like Java and Scala.

Research Gap in Coding Rule Verification

Despite these advancements, a notable gap exists in the application of graph-based approaches specifically for coding rule verification. Existing research has primarily targeted:

- Security vulnerability detection
- Program comprehension and visualization
- Architecture and dependency analysis
- Performance optimization

This gap is particularly relevant for Ada development, where strict coding standards are often mandatory due to safety-critical applications. Our research addresses this gap by applying graph-based analysis specifically to Ada coding rule verification—an application that presents unique challenges:

- Ada's strong typing system and complex semantics require more sophisticated representations
- Safety-critical applications demand higher precision in rule verification
- Industry coding standards for Ada (like SPARK or those used in aerospace) involve complex cross-module rules

By extending graph-based approaches to Ada coding rule verification, our work complements existing research while exploring new territory in the application of graph databases to static analysis. The following chapters will evaluate the effectiveness of this approach in addressing the specific challenges of coding rule verification for Ada programs.

Expressive Rule Definition with Graph Patterns

One of the most compelling advantages of graph-based approaches for coding rule verification is the ability to express rules as intuitive graph patterns. Cypher queries combine a `MATCH` clause that describes the pattern to search for, optional `WHERE` predicates that restrict nodes or relationships based on labels and properties, and a `RETURN` clause that specifies which elements should be reported.

As a simple illustration in our context, consider the Ada procedure `Cat_Laser_Pointer` introduced earlier in this chapter. We can use Cypher to retrieve all subprograms that it calls using the following pattern:

Code 1.2 — Simple Cypher query illustrating call relationships

```
MATCH (p:A_PROCEDURE_DECLARATION { name:
    "Cat_Laser_Pointer" })-[:CALLING]->(callee)
RETURN callee
```

We can decompose this query as follows:

- `(p:A_PROCEDURE_DECLARATION \{ name: "Cat_Laser_Pointer" \}` denotes a node with the label `A_PROCEDURE_DECLARATION` and a `name` property equal to `"Cat_Laser_Pointer"`, representing the procedure of interest.
- `-[:CALLING]->` represents a directed relationship of type `CALLING` from Section 2.1, indicating that `p` calls another subprogram.
- `(callee)` introduces a node variable bound to each subprogram called by `p`.
- `RETURN callee` instructs the database to return all nodes matched as `callee`, that is, all callees of `Cat_Laser_Pointer`.

The same building blocks—nodes with labels and properties, typed relationships, and variables bound by pattern matching—are used to express more sophisticated coding rules. Consider, for example, the rule "Do not use magic numbers in conditions." While traditional AST-based analysis would require complex traversal logic with multiple conditional checks, a graph database approach allows us to express this rule as a concise Cypher query:

Code 1.3 — Example Cypher query to detect magic numbers in conditions

```
MATCH (e:AN_INTEGER_LITERAL|A_REAL_LITERAL)-[:IS_ENCLOSE
    D_IN*]->()-[:IS_ENCLOSED_IN { index:1
    }]->(:AN_IF_PATH|AN_ELSIF_PATH)
RETURN e
ORDER BY e.filename, e.line, e.column
```

This query directly captures the essential structure: literal numbers within conditional statements. The pattern uses:

- Node labels to identify the code constructs (literals, conditional statements)
- The `IS_ENCLOSED_IN` relationships to navigate the syntactic structure
- Index values to specify the precise structural relationship

These examples demonstrate how graph patterns align naturally with how coding rules are conceptualized—as structural patterns to identify in code rather than algorithms to execute. The declarative nature of these patterns makes rules more maintainable, comprehensible, and adaptable to evolving coding standards.

1.3.6 Graph Database Management System Used: Neo4j

In order to efficiently store and query the CPG for static analysis, a suitable GDBMS is required. Rather than developing a custom graph database solution from scratch, which would be a significant undertaking and beyond the scope of this research, we opted to leverage an existing GDBMS. This approach allows us to focus on the graph-based static

analysis methodology itself, while benefiting from the robustness, performance, and query capabilities of a mature graph database system.

The requirements for our GDBMS selection fell into several categories:

Functional Requirements:

- Support for property graphs: The GDBMS should natively support the property graph model, allowing properties (key-value pairs) to be assigned to both nodes and edges [9]
- Powerful query language: An expressive and efficient query language for formulating complex traversals and patterns over the code graph
- Extensibility: Ability to integrate custom algorithms to leverage domain-specific knowledge

Performance Requirements:

- Scalability: Ability to handle large and complex codebases efficiently
- Query optimization: Support for efficient graph traversal and pattern matching

Technical Requirements:

- Operating locally: Support for local installation and operation
- Persistent disk storage: Ability to store data persistently to avoid repeated analysis

Strategic Requirements:

- Open-source: To ensure transparency, enable auditing, and facilitate research collaboration
- Active community: For support, documentation, and long-term viability

Based on these criteria, we chose Neo4j as the GDBMS for this study. Neo4j is a widely-used, mature graph database system that fulfills the above requirements [49]. It provides native support for property graphs through its labeled property graph model.

Cypher, Neo4j's declarative query language, is designed for efficient traversal and pattern matching over graph structures.

It is important to note that while Neo4j was selected for this research, the choice of a specific GDBMS is not a critical factor in the overall approach. The proposed graph-based static analysis methodology is not tied to any particular graph database system. As long as the GDBMS supports property graphs and provides a powerful query language, it can be used to implement the technique effectively.

Chapter 2

Methodology

In this chapter, we present a systematic methodology for investigating the efficacy of graph databases for coding rule verification in Ada programming language. Our approach combines practical implementation with rigorous empirical validation to address the research gap identified in the literature review.

As outlined in Section 1.3.5, a significant research gap exists in exploring the scalability and efficiency of graph databases for coding rule verification in Ada. In our investigation, we adopt a mixed-methods approach, leveraging both theoretical foundations from static code analysis using graph databases and practical implementation with experimental validation. This dual approach is crucial for understanding the inherent trade-offs between different static analysis techniques. While theoretical analysis provides insights into the computational complexity and potential benefits of graph-based methods, practical experimentation is essential for assessing real-world performance and identifying potential bottlenecks. Through this chapter, we aim to bridge this gap by providing a concrete implementation and evaluating its performance against established tools.

At the core of our approach, we represent Ada code as a property graph within a Neo4j graph database. This graph-based representation allows us to capture not only the syntactic structure of the code but also the rich semantic relationships between code elements. We express coding rules, which often involve complex relationships between different parts of the code, as graph pattern matching queries that can be evaluated efficiently using the database's query engine. With this method, we address the scalability challenges faced by traditional static analysis tools when dealing with large codebases, where accessing and processing information scattered across multiple

ASTs can be computationally expensive.

Our methodology proceeds in several key stages:

1. **Defining the Graph Schema:** We keep the nodes of the graph as those present in the AST, without performing any reduction or suppression of AST nodes. Some additional nodes are introduced where necessary, for example to explicitly represent the order of function parameters or to encapsulate an entire package as a single entity. The primary methodological focus is on the careful selection and addition of relationships between these nodes, which enables the graph to capture the complex interactions and dependencies required for advanced code analysis. We provide a detailed description of our chosen relationships in Section 2.1.

2. **Selecting Coding Rules:** We select a set of coding rules for experimental evaluation. We choose these rules to cover a range of common coding patterns and complexities, allowing for a comprehensive assessment of our proposed approach. We detail the selected rules in Section 2.2.

3. **Dataset Preparation:** We curate a diverse benchmark dataset of Ada projects to evaluate the performance and scalability of our method. This dataset includes projects of varying sizes and complexities, reflecting real-world Ada codebases. Beyond serving as an evaluation framework for this research, this comprehensive corpus represents a valuable contribution to the Ada static analysis research community, providing a standardized benchmark that can be reused and extended by future researchers. We describe the dataset preparation process in Section 2.3.

4. **Pre-processing and Database Population:** In this stage, we parse the Ada source code, extract the relevant information, and populate the graph database. A key consideration in our approach is optimizing the pre-processing and population steps to minimize overhead, which is particularly important for achieving good performance on smaller projects. We detail this process in Section 2.4.

5. **Coding Rule Implementation:** We implement the selected coding rules as Cypher queries and describe them in Section 2.5.

6. **Benchmarking:** We benchmark tool metrics on datasets of multiple projects of different sizes. We present our benchmark protocol in Section 2.6, which details the procedures and environment used for all tool evaluations.

7. **Evaluation Metrics:** We evaluate the performance of our graph-based approach against the same rules in AdaControl and GNATcheck [2], two traditional coding verification tools. We detail the evaluation approach and the metrics we use in Section 2.7.

2.1 Created relationships

To capture the semantic information necessary for effective static code analysis, we define several types of relationships between nodes in our graph representation. We design these relationships to express the connections and dependencies among various code elements, enabling efficient querying and pattern matching. Some relationships come from the AST structure, others are inspired by CPG patterns, and several are specific to our approach for Ada code analysis.

There exists a direct dependency between the coding rules we want to implement and the relationships we create in the graph. We select relationships that enable efficient implementation of our target coding rules, while attempting to create generic relationships that could support multiple rules. As we add more rules, the graph naturally evolves to become richer with additional semantic connections.

Beyond the rules implemented in this study, we design the schema so that additional families of coding rules can be supported by introducing new relationship types, for instance to capture more precise data-flow or control-flow properties. Each new relationship family affects the construction of the graph: it increases the amount of information that we compute during preprocessing and store in the database, but it can significantly simplify the queries used to express those rules.

The key relationships we create in the graph include:

- **CALLING**: Derived from CG. Connects a subprogram node to the subprograms it calls, allowing for call graph traversal.
- **CORRESPONDING_ACTUAL_ PARAMETER**: Specific to my approach. Links a parameter association node to the corresponding actual parameter value.
- **CORRESPONDING_ASSIGNATION**: Related to data flow analysis but specific to my implementation. Connects a variable node to the corresponding assignment statement node.
- **CORRESPONDING_FIRST_SUBTYPE**: Specific to my approach for type analysis in Ada. Relates a subtype node to its corresponding first subtype declaration.
- **CORRESPONDING_FORMAL_NAME**: Specific to my approach. Links a parameter association node to the corresponding formal parameter identifier.

- **CORRESPONDING_INSTANTIATION**: Specific to my approach for generic instantiation analysis. Connects a node to the corresponding generic instantiation declaration.

- **CORRESPONDING_NAME_DEFINITION**: Inspired by symbol tables but implemented in my graph representation. Relates an identifier, operator symbol, character literal, or enumeration literal node to its corresponding definition.

- **CORRESPONDING_PARAMETER_ SPECIFICATION**: Specific to my approach for parameter analysis. Links a parameter association node to its corresponding parameter specification.

- **CORRESPONDING_ROOT_TYPE**: Specific to my approach for type hierarchy analysis in Ada. Connects a derived type node to its original ancestor type.

- **CORRESPONDING_SPECIFICATION**: Specific to my approach. Relates a subprogram, package, task body declaration, or expression function declaration node to its corresponding specification, if available.

- **CORRESPONDING_TYPE_ DECLARATION_VIEW**: Specific to my approach for type analysis. Links a type declaration node to its corresponding definition characteristics.

- **IS_ANCESTOR_OF**: Derived from type hierarchy analysis but implemented in my specific graph representation. Expresses the ancestor-descendant relationship between type declarations.

- **IS_ENCLOSED_IN**: Directly derived from AST structure. Represents the parent-child relationship between nodes. This is the only relationship found in an AST.

- **IS_OF_TYPE**: Inspired by symbol tables but implemented in my graph representation. Connects a declaration node (e.g., component, constant, discriminant specification) to its corresponding type definition.

- **IS_PROGENITOR_OF**: Specific to my approach for interface inheritance analysis in Ada. Expresses the relationship between an interface type and its directly implementing types.

We carefully choose these relationships to strike a balance between expressiveness and database efficiency. While adding more relationship types can increase the semantic richness of the graph, it also affects several aspects of the database:

- **Query complexity:** More relationship types can make queries more complex to write and maintain, as they need to account for various possible connections between nodes.

- **Creation time:** Calculating and establishing each relationship type requires processing time during database population.

In practice, the selection of relationships is primarily guided by the families of coding rules we want to support: adding a relationship can make certain rules straightforward to express but may increase population time and overall database size; conversely, omitting it keeps population leaner but can lead to more complex queries.

We select relationships based on their semantic significance, particularly how they facilitate the enforcement of specific coding rules. Not all coding rules benefit equally from these relationships, but we choose ones that are instrumental for rules critical to assessing code quality and security. This targeted approach maximizes the utility of the database while maintaining manageable size and performance.

For calculating these relationships, we primarily rely on multiple traversals of the AST during the preprocessing phase, before database population. Most relationships are computed by analyzing the AST structure and creating appropriate connections between nodes. However, we calculate certain relationships directly within the graph database after the initial AST nodes are imported:

- The **CALLING** relationships (CG) are computed by executing specific queries on the populated graph to identify function calls and their targets.

- Ancestor-related relationships (**IS_PROGENITOR_OF** and **IS_ANCESTOR_OF**) are also calculated directly in the graph database after the initial node population.

Currently, we create all of the above relationships regardless of whether they are used by the specific coding rules being applied. An alternative approach would be to create only the relationships required by the coding rules specified by the user. However, we believe the better long-term strategy is to build a comprehensive graph during the initial pre-processing phase. This approach ensures that all potentially useful relationships are available for analysis without the need to re-process the entire codebase when new rules are introduced. Furthermore, when the source code evolves, the graph database can be updated incrementally by modifying only the nodes and relationships affected by the changes, rather than clearing and repopulating the entire database. Creating

only a subset of relationships would reduce initial processing time but would require re-preprocessing the entire codebase when new rules need different relationships.

At present, our implementation rebuilds the entire graph from scratch whenever the source code changes, rather than attempting to incrementally update only the affected nodes and relationships. While incremental updates (where only the parts of the graph impacted by code modifications are updated) represent a promising direction for future work, they have not yet been implemented or evaluated in this study. For the purposes of this research, we focused on the construction and analysis of a comprehensive graph generated in a single pre-processing phase. Throughout this work, we carefully selected and refined the relationship types to ensure they provide maximum value for static analysis queries.

2.2 Selection of coding rules

To evaluate our graph-based static analysis methodology, we selected a set of coding rules. We chose these rules to evaluate a variety of code constructs and patterns that traditional Ada analysis tools examine, enabling direct comparison of results between our proposed approach and established tools.

We aligned our selected rules with those found in the widely-used Ada static analysis tools AdaControl and GNATcheck. This alignment facilitates direct comparison of findings to help validate our graph database implementation. In spirit, we aim to support the same families of checks as AdaControl (cf. the AdaControl user guide[1]), although purely stylistic rules are not our priority in this work. Our focus is on rules that verify the usage and non-usage of language features and code constructs. As an illustration of our target scope, we are interested in rules that help verify compliance with the Ravenscar profile. For this initial proof of concept, we primarily focused on rules that were straightforward to implement, while still representing different analysis categories.

In selecting these particular coding rules, we considered several practical factors. Given the time constraints of our research, we prioritized rules that were feasible to implement within the research timeline while still enabling meaningful comparisons with existing tools. We acknowledge that this selection is not exhaustive and serves primarily as a proof of concept.

It is important to note that the rules that would most dramatically demonstrate the

[1] `https://www.adalog.fr/compo/adacontrol_ug.html`

advantages of our graph-based approach are often those requiring complex, cross-unit analysis and deep AST exploration—precisely the types of rules that might be difficult or inefficient to implement in traditional tools due to their architecture. Such complex rules would likely benefit most from the graph database's capability to efficiently traverse relationships and correlate information across the codebase.

The following complex rules, which are currently implemented only in AdaControl (not in GNATcheck), would be valuable additions in future work to further demonstrate the scalability advantages of the graph-based approach:

- **Abnormal_Function_Return:** Controls functions that may not terminate normally, i.e. where `Program_Error` could be raised due to reaching the end of the function without encountering a `return` statement.
- **Directly_Accessed_Globals:** Controls that global variables in package bodies are accessed only through dedicated subprograms (through getters / setters). Especially, it can be used to prevent race conditions in multi-tasking programs.
- **Movable_Accept_Statements:** Controls statements that are inside accept statements and could safely be moved outside. This is for optimizing performance in multi-tasking systems.
- **No_Operator_Usage:** Controls integer types that do not use any arithmetic operators, which indicates that they might be replaceable with other kinds of types.
- **Complete Variable_Usage:** A full implementation of the variable usage tracking, described below

We did analyze one more complex rule, **Variable_Usage**, which tracks read and write operations on variables, though we only implemented it partially as a case study.

We categorize these rules into three types based on their analysis scope:

- **Local rules:** Rules that require analysis within a single compilation unit
- **Intermediate rules:** Rules that primarily analyze a single unit but may require information from other units
- **Global rules:** Rules that necessarily analyze relationships across multiple compilation units

We implemented the following coding rules:

- **Local rules:**
  - **Abort_Statements:** Identifies uses of the `abort;` statement. This could be achieved with simple text matching (like `grep`). This check is crucial for ensuring compliance with the Ravenscar profile, which prohibits abort statements to guarantee deterministic behavior in real-time systems.
  - **Blocks:** Reports block statements
  - **Constructors:** Identifies primitive functions of tagged types that have a controlling result but no controlling parameter, focusing on potential object construction patterns.
  - **Enumeration_Representation_Clauses:** Reports enumeration representation clauses (like `for Enum use (10, 20, 30);`) to identify potential portability issues.
  - **Renamings:** Detects all renaming declarations to track alternate names for entities. Many coding standards restrict the use of renamings, either prohibiting them entirely or allowing only specific cases (such as package renamings while forbidding variable renamings) to maintain code clarity and prevent indirect references that could complicate maintenance.
  - **Slices:** Identifies array slice operations to monitor potential performance impacts.
- **Intermediate rules:**
  - **Abstract_Type_Declarations:** Identifies declarations of abstract types, including those in generic formal parts, requiring potential cross-unit analysis for full type information.
- **Global rules:**
  - **Too_Many_Parents:** Analyzes inheritance hierarchies across compilation units, identifying tagged types, interface types, tasks, or protected types exceeding a specified number of parents through derivation or interface implementation. Various programming standards limit the number of ancestors a type can have to prevent overly complex inheritance hierarchies that could make the code difficult to understand and maintain.
  - **Variable_Usage:**[1] Tracks read and write operations on variables throughout the program, including through procedure calls and renamings across different units.

[1] Partially implemented

The selected rules demonstrate different scenarios where a graph-based approach offers advantages over traditional analysis methods:

- For seemingly simple rules like **Abort_Statements**, while a text-based approach (e.g., `grep`) might appear sufficient, such methods lack semantic understanding. For example, a text search would report occurrences of `abort` in comments, string literals, or within identifiers like `abort_handler`. Our graph approach significantly improves accuracy by analyzing the syntactic and semantic structure of the code, distinguishing between actual `abort` statements and other textual occurrences. While no static analysis approach can claim perfect accuracy for all possible programs (due to theoretical limitations of static analysis), working at the AST level provides a more reliable foundation than text-based pattern matching.

- For rules requiring AST traversal like **Abstract_Type_Declarations**, traditional analyzers must repeatedly traverse the entire AST to locate all abstract type declarations and verify their properties. In contrast, our graph database allows direct access (in $O(1)$) through indexed labels for immediate retrieval of type declarations, combined with efficient relationship traversal to check abstraction properties.

- For more complex rules like **Constructors** or **Too_Many_Parents**, our approach can be beneficial as it can efficiently correlate information from multiple AST locations that would otherwise require complex visitor patterns or multiple passes in traditional analysis.

We emphasize that the data we extract and store in the database constitutes a fully elaborated AST, augmented with additional relationships as detailed in section 2.1. Currently, we extract all information for all nodes and relationships of the AST, rather than only extracting information required by specific coding rules. This comprehensive approach provides more flexibility but comes with increased storage requirements.

2.3 Selection and Preparation of the Benchmark Dataset

To comprehensively evaluate our proposed techniques, we needed an appropriate Ada codebase as a benchmark dataset. Our key considerations for dataset selection were:

- Representative of real-world industrial code: This enables us to assess scalability and performance on complex logic.
- Wide variety of size and complexity: Required to stress test our approach on a large variety of cases.
- Ability to compile on Linux x86_64: To match our target benchmark architecture.
- Limited to Ada 2012 code: Due to our tooling (ASIS) constraints.

To obtain a representative real-world Ada codebase, we utilized the Alire [6] package manager[1] to download libraries from their repository. Alire provided us access to a comprehensive collection of open source Ada projects. We downloaded all available libraries from the Alire repository. In addition, we manually added Ada GitHub projects that matched our requirements (containing `alire.toml`, compiling on Linux x86_64, and limited to Ada 2012 code).

To avoid duplicating dependent libraries, we created local links between projects. For example, if a project A depends on a project B, we edited the dependency of project A to point to the local (downloaded) version of project B. This resulted in a self-contained repository [4] with all dependencies encapsulated to ensure reproducibility.

To accomplish this task, we developed a set of scripts to edit `alire.toml` project files to create the local link to each dependency.

After downloading repositories, we applied filters to only keep projects that compile without generating compiler errors, ASIS errors, or SCA tool errors when running SCAs tools.

In our selection, we also excluded the libadalang project. While this project met our previous requirements (compiles without errors), its particular design, with a file containing more than 100 000 Lines of Code (LoC), created a huge tree-swapping issue that produced insignificant results. Since our approach currently relies on ASIS for the pre-processing phase[2], it would take several days to complete it.

In total, we included 134 successfully compiling projects in our benchmark, encompassing 2 643 887 lines of Ada code[3]. This corpus provides us with extensive coverage across a diverse set of coding styles, complexities, domains and sizes, ranging from small (100s LoC) to large (+100 000 LoC) codebases.

[1] It is the equivalent of PIP for Python, Crates for Rust or NPM for Node.js

[2] As described in Section 2.4

[3] Counted using SCC [43], excluding blank and comment lines

We categorized projects based on LoC to analyze scaling behavior:

- Small: 0-10 000 LoC
- Medium: 10 001-30 000 LoC
- Large: more than 30 000 LoC

This categorization enables us to analyze how each tool's performance scales with project size and complexity.

By leveraging Alire and applying careful filtering criteria, we created a sizable benchmark suite to facilitate thorough comparative assessments between our graph-based approach and conventional analysis techniques.

2.3.1 Dataset Characteristics and Potential Bias

Our corpus was assembled by downloading all available Alire packages at a specific point in time and then filtering projects to keep only those that successfully compile on our target environment and for which the static analysis tools run without errors. This pragmatic approach ensured reproducibility and a broad coverage of Ada code, but it also introduces possible selection biases:

- Projects that fail to compile or that trigger tool errors are excluded, potentially removing difficult edge cases.
- The dataset reflects the state of open-source Ada code in Alire at that specific snapshot, which may not fully represent closed-source industrial projects.
- The focus on Ada 2012 compatibility (due to tooling constraints, see Section 2.6) may reduce exposure to newer Ada 2022 constructs.

2.3.2 Language Feature Usage in the Dataset

To ensure our benchmark allows for a comprehensive evaluation of static analysis tools, it is crucial that the dataset covers a wide spectrum of Ada language features. An analysis of language features used and the imports used across the selected projects reveals a rich diversity of usage patterns, confirming that the dataset is not limited to a simple subset of the language but represents real-world usage.

Standard Library and Data Structures: A significant portion of the projects makes extensive use of the Ada standard library. We observe frequent usage of:

- **Containers:** Usage of `Ada.Containers` including vectors, maps, and linked lists (both definite and indefinite forms), as well as sorting algorithms.
- **String Handling:** Extensive use of `Ada.Strings`, including unbounded strings and UTF encoding support.
- **Input/Output:** Standard text and stream I/O, as well as specialized I/O for complex types.

Generics and Polymorphism: The dataset includes projects that heavily rely on generic programming, not only through standard containers but also via custom generic units. This provides a robust test bed for analyzing instantiation chains and parametric polymorphism.

System and Low-Level Programming: Reflecting Ada's strong presence in systems programming, the dataset contains projects using:

- `Ada.System` hierarchy (e.g., `Ada.System.Address`, `Ada.System.Storag` ⌋ `e_Pools`).
- Interfacing with C (`Interfaces.C`).
- GNAT-specific extensions (e.g., `Gnat.Sockets`, `Gnat.Expect`).

Concurrency: While less dominant than sequential logic, the dataset does include projects utilizing Ada's concurrency features, such as `Ada.Task_Identification` and synchronized queue interfaces, allowing us to verify that our tools handle multi-tasking constructs correctly.

Quantitative Coverage: To provide concrete evidence of feature diversity, Table 2.1 summarizes the usage of major Ada language constructs across our dataset. The table groups related features by category and reports both the number of projects using each feature and the total number of occurrences. This demonstrates that our benchmark exercises a substantial portion of the Ada language, including advanced features such as parametric polymorphism (generics), object-oriented programming (tagged types with inheritance), exception handling, and concurrency primitives. Complete detailed metrics for all language features are available in Appendix C.1.

Table 2.1: Ada Language Feature Coverage in the Benchmark Dataset

Feature Category	Projects	Total Uses
Parametric Polymorphism (Generics)		
Generic units declared	59	640
Generic instantiations (public)	47	992
Generic instantiations (private)	40	256
Generic instantiations (local)	51	553
Object-Oriented Programming (Tagged Types)		
Tagged type hierarchies (max depth 7)	69	1304
Abstract type declarations	25	166
Types with discriminants	55	397
Controlled types	29	275
Exception Handling		
Exception declarations	53	482
Raise statements (all)	77	2889
Exception handlers (others)	44	1402
Locally handled exceptions	8	23
Concurrency and Tasking		
Task declarations	3	6
Protected object declarations	9	22
Accept statements	9	50
Entry call statements	12	113
Delay statements (relative)	12	69
Access Types and Aliasing		
`'Access` attribute	54	8302
`'Unchecked_Access` attribute	21	560
`'Address` attribute	36	1145
Access-to-subprogram types	34	1319
Parameter aliasing (possible)	3	4
Advanced Features		
Operator overloading	36	473
Representation clauses	43	469
`Unchecked_Conversion` (Address→Access)	9	869

The quantitative analysis reveals both strengths and limitations of our dataset. On the positive side, we observe strong coverage of core Ada features: generics are widely used (59 projects declaring generic units, with 992 public instantiations), object-oriented programming is well-represented (69 projects using tagged type hierarchies with inheritance depths reaching up to 7 levels), and exception handling is present (77 projects with raise statements totaling 2889 occurrences, of which 23 are locally handled within the same procedure in 8 projects). Access type manipulation, a critical feature for systems programming, shows extensive usage with 8302 uses of the `'Access` attribute across 54 projects.

However, the dataset shows limited coverage of concurrency features. Task declarations appear in only 3 projects (6 total instances), protected objects in 9 projects (22 instances), and tasking primitives such as accept statements (9 projects, 50 uses) and entry calls (12 projects, 113 uses) remain relatively scarce. Similarly, parameter aliasing—a subtle but important aspect of Ada semantics—appears in only 3 projects with 4 possible aliasing situations detected. This limited representation of concurrency constructs reflects the dominance of sequential logic in the open-source Ada ecosystem available through Alire, rather than a deliberate filtering choice.

These gaps represent opportunities for future dataset improvement. Specifically, we could enhance coverage by:

- Actively seeking projects from domains where concurrency is central (e.g., real-time systems, embedded controllers, or distributed applications),
- Including curated industrial code samples when available under permissive licenses,
- Developing synthetic benchmarks that specifically exercise underrepresented language features.

Such augmentation would enable more comprehensive evaluation of static analysis tools' capabilities across the full spectrum of Ada constructs, particularly for features critical in safety-critical and real-time domains where Ada is heavily deployed.

This variety ensures that our performance results reflect the behavior of static analysis tools when observing the full "treated language" as found in diverse, open-source ecosystems.

2.4 Pre-processing and database population

2.4.1 Pre-processing

To reduce the development time, we started from the source code of AdaControl [5], which is built upon the ASIS [42] library to construct ASTs.

As we explained in section 1.2.3, ASIS has a performance issue (TreeSwapping) and is strongly coupled with the AdaCore's GNAT compiler ecosystem.

> ⚠ **Warning 2.1 — Tree Swapping issue in our approach**
>
> It is important to note that since we currently use ASIS for parsing the Ada source code during the data collection phase, there is a Tree Swapping issue. However, we emphasize that this Tree Swapping issue is only relevant during the data collection phase. There is no Tree Swapping issue in our database population phase, nor in controlling coding rules.

Although ASIS has played a crucial role in our research as a proof of concept, we plan to transition to the newer library libadalang [3] for future endeavors. Libadalang is a semantic engine for Ada that provides a high-level interface to analyze Ada sources. It is designed to be more efficient and flexible than ASIS, supporting newer Ada standards (Ada 2022) and offering a more modern API. At the time of our initial research, libadalang was not yet fully mature to support our specific requirements for static code analysis. It did not support all the language features, and therefore, did not allow us to support the complexity of coding rules we needed.

2.4.2 Database population

To enable our graph-based analysis, we made deep modifications to AdaControl, replacing the rule checking system with a mechanism to populate the Neo4j database.

In our code, we employed a Producer-Consumer pattern, where the producer is the AST traversal task and the consumer is the query generation task. During the AST traver-

sal, our producer task visits each node in the abstract syntax tree and collects relevant syntactic and semantic information about the code. We then store this information in a shared data structure called a protected object, which ensures safe access from multiple concurrent tasks.

Our use of a protected object allows for efficient communication between the producer and consumer tasks, minimizing the need for synchronization and enabling parallel processing. This design was particularly easy to implement in Ada.

Our consumer task, running concurrently with the producer, reads the information from the protected object as it becomes available and processes this information to generate a series of Cypher queries.

The following subsection describes our first implementation and the optimizations we made to make the database population efficient.

Optimizing Database Population

In our initial approach to database population, the Consumer sent an HyperText Transfer Protocol (HTTP) request immediately upon obtaining data from the protected object. With this methodology, our analysis of a test file containing 3,876 LoC required more than one hour to populate the database. Consequently, addressing this inefficiency became critical for us to process large volumes of code.

We centered our reflection on refactoring the export system to make it more flexible and modular. We applied the Observer design pattern to our Consumer task, which enabled us to wire multiple export systems and simplified the addition of new ones. We also included a JavaScript Object Notation (JSON) Lines[1] export of our internal representation, which allowed us to doing prototype using a scripting language.

Subsequently, we implemented a Node.js prototype for database population. Through our study of Neo4j's API, we learned to optimize the query construction for repeated query scenarios by replacing literals with parameters to leverage server-side caching of query plans [24]. The code presented in 2.2 demonstrates an illustration of a node creation query utilizing literals, whereas 2.3 exhibits the identical query employing parameters.

[1] More information on the official website: `https://jsonlines.org/`

Code 2.2 — Example of Cypher query using literals

```
CREATE(n:Person {_id:"id1",name:"Keanu Reeves"});
CREATE(n:Person {_id:"id2",name:"Carrie-Anne Moss"});
```

Code 2.3 — Example of Cypher query using parameters

```
// Parameters of the query
:params rows ⇒ [{"_id":"id1","name":"Keanu
    Reeves"},{"_id":"id2","name":"Carrie-Anne Moss"}]
// The query
UNWIND $rows AS row
CREATE(n: `UNIQUE IMPORT LABEL`{node_id: row._id})
SET n += row.properties
SET n:Person;
```

Our utilization of parameters, alongside the addition of properties and labels to the node through the SET operation, significantly enhanced the performance of populating the database.

Note 2.4 — Database specific optimization

We want to emphasize that the parametrized query's optimization varies depending on the specific GDBMS employed. Similar optimizations may need to be implemented on a case-by-case basis, contingent upon the chosen GDBMS.

This functional prototype informed our implementation in Ada.

However, we encountered two major issues. Firstly, a single file containing all nodes, relationships and queries exceeded stack limits on large codebases. To remedy this, we implemented a file management system that distributes queries across multiple output files based on node types and relationships. Each file has a maximum size limit determined by OS stack constraints, Ada type limitations, and average query size.

This organization is not a pipeline that allows immediate database population while parsing continues. Rather, it works as follows: as nodes are encountered during parsing, they are written to type-specific files (e.g., all procedure nodes to one file, all package nodes to another); when a file reaches its size limit, a new file of the same type is created.

Similarly, relationships are written to dedicated relationship files. Due to referential integrity requirements, all node files must be completely processed before relationship files can be loaded into the database, otherwise the database would create empty placeholder nodes or even duplicates when encountering relationships referencing not-yet-loaded nodes.

Secondly, reading the file stream to send HTTP requests to Neo4j's API also caused stack overflows. We addressed this by allocating the stream through a pointer to utilize the heap rather than the stack.

> **Information 2.5 — HTTP request streaming issue**
>
> During early prototypes, streaming large batches over HTTP from a stack-allocated stream led to stack overflows. Switching the stream allocation to the heap resolved the issue without affecting the population pipeline semantics described above.

Our efforts successfully optimized population time. On a 3876 lines test file, we reduced the time from over an hour to approximately 30 s. We also added the entirety of AdaControl (with dependencies), representing around 100 000 lines and created a large graph in under 2 min.

Query structure of our current approach

We organize our new query structure into multiple JSON files, which we categorize as follows:

- `0_XXX`: Database initialization
- `1_XXX`: Node creation
- `2_XXX`: Relationship creation
- `3_XXX`: Database finalization

Our initialization queries solely establish custom labels and indices used for populating the database, while our finalization queries encompass the following steps:

1. Creating an order with relations to the parent

2. Generating the call graph
3. Establishing ancestor relationships
4. Removing labels employed in the database population
5. Removing constraints employed in the database population

Unlike most other relationships which we calculate during the AST traversal phase, we compute the CG and ancestor relationships directly in the database after populating it with the basic structure and initial relationships. This approach leverages the graph database's querying capabilities for these particular relationships, which are more efficiently calculated by analyzing the complete graph structure rather than during the initial AST traversal.

Information 2.6 — On the timing of relationship computation

It is important to clarify how the time spent computing the call graph and ancestor relationships is accounted for in our methodology. In our current implementation, this computation is included in the database population phase, rather than in the rule evaluation phase. This differs from traditional static analysis tools, where equivalent computations are typically performed each time rules are evaluated and thus are counted within the rule-checking time.
Our approach aims to integrate as much semantic information as possible directly into the graph during population. This design choice avoids redundant computation for each rule evaluation, making the process more efficient for repeated analyses.
Ideally, the workflow could be decomposed into three distinct phases:

1. Population of the database with initial syntactic elements
2. Enrichment of the database with advanced relationships (such as call graph and ancestry)
3. Rule verification

For a perfectly fair comparison with tools that do not use a graph database, one could compare the sum of phases 2 and 3 with the total analysis time of non-graph-based tools. However, in our current setup, phases 1 and 2 are performed together for efficiency, and we did not anticipate separating their timings. This could be explored in future work, as splitting these phases might slightly increase the total population time compared to the current combined approach.

After we complete the exploration of the entire code, we sequentially read the files to populate the database (utilizing an HTTP connection), except for those starting with ”1” or ”2,” which we execute concurrently. Specifically, we first execute all files beginning with ”1,” followed by those starting with ”2”. The diagram in Figure 2.1 illustrates our process.

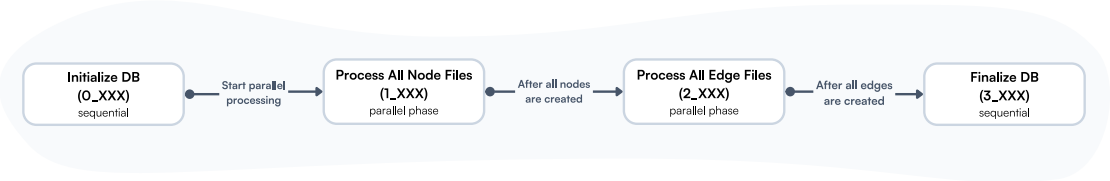


Figure 2.1: How DB population works

As previously said, this approach yields 100-fold improvement in the time required for the database population compared to ”real-time feeding”.

Information 2.7 — Database parallel population

We want to emphasize that it was possible to feed the database in parallel with Neo4j. This is not necessarily the case with all GDBMS.

Conclusion on database population

Through this robust population process, the syntactic AST is efficiently transformed into a semantically rich populated graph representation to enable querying and analysis. The decoupled architecture and parallelized ingestion were key to achieving scalable graph construction.

2.5 Development of Coding Rules

We developed our coding rules to align with those of existing Ada static analysis tools to facilitate comparison of results. We implemented these rules using Cypher query language, which is the only query language supported by Neo4j. In this section, we explain the advantages of using Cypher for rule development and provide examples demonstrating its expressivity compared to traditional approaches.

We provide a concise overview of graph query languages, including Cypher, in Section 1.3.4 of the literature review. In practice, Cypher's declarative pattern-matching is well suited for expressing structural rules; however, expressing comprehensive semantics for complex rules can lead to long queries when many language cases must be covered explicitly. Where appropriate, we decompose such queries or combine Cypher with post-processing, as illustrated later in this section.

2.5.1 Example Of Cypher Query

Code 2.8 shows our Cypher query related to the "Too Many Parents" coding rule, which checks for type declarations with an excessive number of ancestor types.

Code 2.8 — Example Cypher query for the "Too Many Parents" coding rule

```
// Parameters of the query
:params { "minNbParents": 2 }

// The query
MATCH (typeDecl)<-[r:IS_PROGENITOR_OF|IS_ANCESTOR_OF]-(parent)
WITH typeDecl, count(r) as nbParents
WHERE nbParents ⩾ $minNbParents
RETURN typeDecl, nbParents
ORDER BY typeDecl.filename, typeDecl.line, typeDecl.column
```

This query retrieves type declaration nodes (`typeDecl`) having at least two incoming relationships labeled `IS_PROGENITOR_OF` or `IS_ANCESTOR_OF` from parent nodes, as specified by the `minNbParents` parameter. We order the results by the filename, line, and column properties of each `typeDecl` node.

2.5.2 Common Query Structure

We share the structure shown in Code 2.9 across the following coding rules:

- Abort Statements

- Blocks
- Enumeration Representation Clause
- Slice

Code 2.9 — Common structure for simple Cypher queries

```
MATCH (e:<a-set-of-labels>)
RETURN e
ORDER BY e.filename, e.line, e.column
```

This query matches nodes with certain label(s) (`<a-set-of-labels>`) and returns them ordered by file position. For example, in our "Abort Statements" rule, we only consider nodes with the `AN_ABORT_STATEMENT` label. In our "Renamings" rule, we match multiple labels as shown in Code 2.10.

The simplicity of these queries demonstrates another advantage of our graph-based approach. While a text-based approach like grep could potentially identify these patterns, our method provides semantic understanding rather than purely textual matching. Our graph representation ensures we're identifying actual code constructs rather than matching text patterns that might appear in comments or strings, and it allows us to accurately locate each occurrence within its proper context in the code structure.

Code 2.10 — Cypher query for the "Renamings" rule

```
MATCH (e:AN_OBJECT_RENAMING_DECLARATION|
A_PACKAGE_RENAMING_DECLARATION|
A_FUNCTION_RENAMING_DECLARATION|
A_PROCEDURE_RENAMING_DECLARATION|
AN_EXCEPTION_RENAMING_DECLARATION|
A_GENERIC_PACKAGE_RENAMING_DECLARATION)
RETURN e
ORDER BY e.filename, e.line, e.column
```

2.5.3 Complex query example

As we observed, controlling the presence of simple constructs in the code is relatively straightforward. However, as we demonstrate next, the complexity of the query can increase proportionally to the inherent complexity of the rule being implemented.

Let us go back to the ”Constructors” rule (previously introduced in Section 2.2) which aims to identify constructor functions in object-oriented Ada code. A constructor in this context is a function that returns a tagged object type without taking that same type as a parameter.

Code 2.11 — Cypher query for the “Constructors” rule

```
// First, we match every function decl that returns a tagged type
MATCH
  (function:A_FUNCTION_DECLARATION|
   A_FUNCTION_RENAMING_DECLARATION|
   AN_EXPRESSION_FUNCTION_DECLARATION)
  <-[:IS_ENCLOSED_IN]-(:AN_IDENTIFIER)
  -[:CORRESPONDING_NAME_DEFINITION]->()
  -[:IS_ENCLOSED_IN]->(typeDef)
  -[:CORRESPONDING_TYPE_DECLARATION_VIEW]->
  (:A_TAGGED_RECORD_TYPE_DEFINITION|
   A_TAGGED_PRIVATE_TYPE_DEFINITION|
   A_TAGGED_INCOMPLETE_TYPE_DECLARATION|
   A_DERIVED_RECORD_EXTENSION_DEFINITION)

// Get the type of each parameter of the function, if exists
OPTIONAL MATCH
  (function)<-[:IS_ENCLOSED_IN]
  -(:A_PARAMETER_SPECIFICATION)<-[:IS_ENCLOSED_IN]
  -(:AN_IDENTIFIER)
  -[:CORRESPONDING_NAME_DEFINITION]->()
  -[:IS_ENCLOSED_IN]->(parmType:A_DECLARATION)

// Collect parameters and filter functions that don't take their return type as parameter
```

```
24 WITH collect(parmType) AS functionParams, function,
       typeDef
25 MATCH (function)
26 WHERE NOT apoc.coll.contains(functionParams, typeDef)
27
28 // Return matching functions ordered by file location
29 RETURN function
30 ORDER BY function.filename, function.line,
       function.column
```

Figure 2.2 illustrates the pattern used in the Cypher query to identify constructors. The diagram is color-coded to enhance readability, with each color representing a distinct relationship path in the graph:

- The **vertical structure** shows two main branches: a left branch (in blue) that traces how a function returns a tagged type, and a right branch (in yellow) that identifies parameter types.
- The **green relationships** connect identifiers to their containing function, showing how function names and return types are associated.
- The **red relationships** connect parameter specifications to their function and identify parameter names.
- The **blue and yellow paths** trace through name definitions to their respective type definitions.
- At the top, the **elliptical constraint** (typeDef $\notin$ functionParams) connected by a **purple "Apply filter"** arrow enforces that no parameter type matches the return type—the defining characteristic of a constructor.

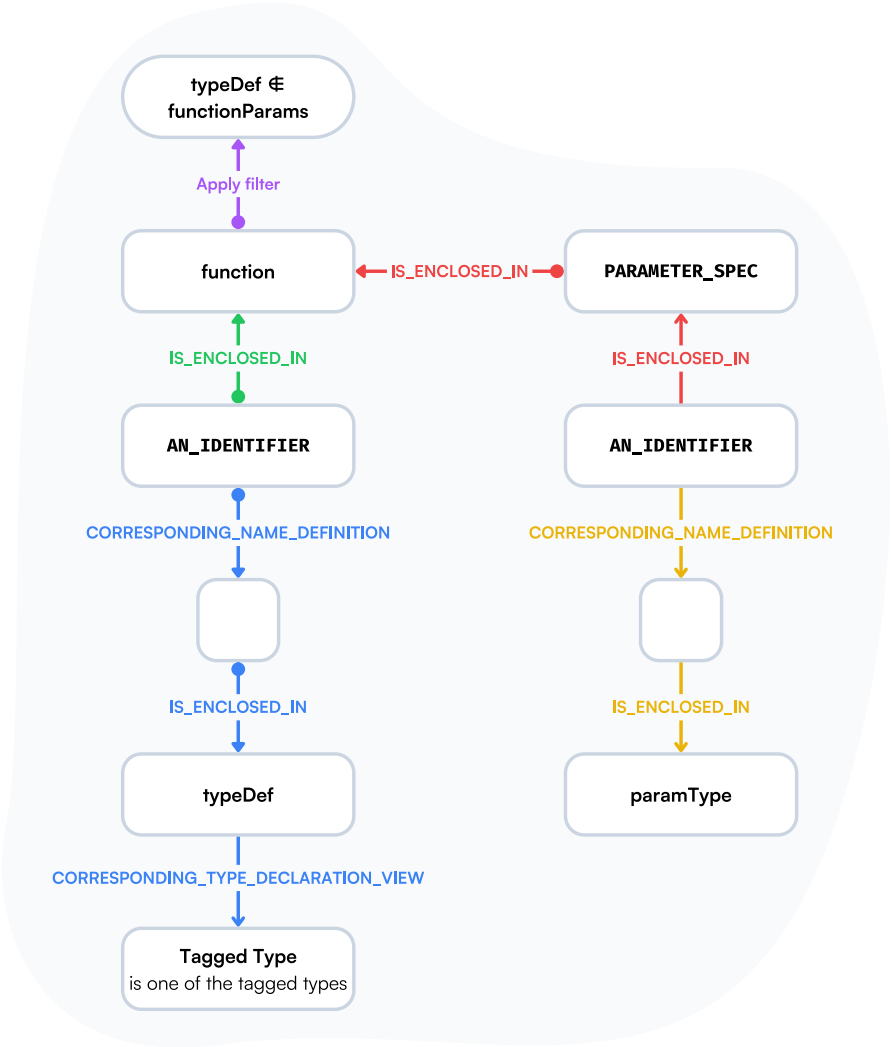


Figure 2.2: Graphical representation of the ”Constructors” rule query pattern

The rectangular nodes represent elements from the Ada AST as stored in the graph database, organized by their role in the code structure. The elliptical node at the top represents a constraint that is applied during query execution. Each edge is explicitly labeled with its relationship type (e.g., `IS_ENCLOSED_IN`, `CORRESPONDING_NAME_DEFINITION`) exactly as it appears in the graph database, making it possible to directly trace how the abstract syntax tree relationships are traversed in the query.

In plain language, this query performs the following steps:

1. Identifies all functions (including function declarations, renaming declarations,

and expression functions) that return a tagged type (like tagged records or private types)

2. For each of these functions, collects all parameter types
3. Filters out functions where any parameter is of the same type as the return type
4. Returns the remaining functions, which are constructors that create a tagged object without taking that same type as input

Figure 2.2 provides a graphical representation of this query pattern, showing how the nodes and relationships are connected.

This query demonstrates the expressivity of Cypher for complex pattern matching. In a traditional imperative approach, we would need to write code to:

1. Iterate through all function declarations
2. For each function, determine its return type
3. Check if the return type is a tagged type
4. Gather all parameter types for the function
5. Check that none of the parameter types match the return type

Each of these steps would require multiple lines of code, traversal of the AST, and management of intermediate collections. Our Cypher implementation encapsulates this logic in a single query that directly expresses the pattern we're looking for. The database engine handles the complexity of finding these patterns efficiently across the entire codebase.

Cypher follows a declarative, pattern-matching paradigm rather than an imperative one. We found this makes it challenging to execute iterative, procedural operations commonly encountered in static analysis. In the next section, we show an example of issues we encountered with Cypher on a complex query.

2.5.4 Analysis of Variables Usage

We wrote an analysis rule to determine: For each variable in a program, is the variable read and/or written?

This coding rule provides valuable insights into variable usage within the software. However, our initial attempt to implement this rule using a single Cypher query resulted in prohibitive performance penalties, with the query timing out on even moderately sized codebases of approximately 8000 LoC. Our query required more than 200 lines of Cypher and did not manage the case of `renames`.

Our further investigation revealed the likely cause to be the formulation of the query, which appeared to induce multiple cartesian products that substantially increased processing time. To address this inefficiency, we reformulated the query into two separate parts:

- Our first query focuses exclusively on normal and generic variable usage contexts, excluding instantiations of generics.
- Our second query handles only instantiations of generics.

This split query approach yielded improved performance, avoiding timeouts on intermediate codebases. On our test case, we reduced the analysis time from 3.6 s to 1.92 s with the dual query technique - an improvement of 1.875 × faster. We include the full code in appendix B.1.

The complete implementation of these queries, available in Appendix B.1, demonstrates both the power and limitations of using Cypher for complex static analysis tasks. The declarative nature of Cypher allows us to express complex patterns concisely, but it can also lead to performance challenges when dealing with very complex patterns that would benefit from procedural control flow.

However, our rewritten rule does not yet fully support all cases. A key outstanding case is that of renamings, which allow a variable to be referenced through alternate names. We found that renamings can obscure the underlying variables being read or written, presenting challenges for accurate analysis. For example, the code `Tab (I) := My_Var` has two read variables (`I`, `My_Var`) and one variable written, `Tab`. In the case of renamings, we can do renaming like this `My_Renamings renamings Tab (I)` and use it like `My_Renamings := My_Var`. Therefore, the variable `Tab` (written) and `I` (read) are hidden but are still counted as read and written.

Due to the complexity of comprehensively handling all renaming cases, we postponed the support for this coding rule. Nonetheless, our exploration of this rule provided us with valuable insights that extend beyond this specific case. It shed particular light on the critical importance of query formulation in determining query efficiency within graph databases.

We observed that even small changes in how a query is expressed can dramatically impact its performance characteristics. The cartesian product issue we encountered is just one example of a broader class of performance concerns that arise when working with declarative query languages. It is somewhat surprising that, in a graph database system, query optimization does not automatically handle these performance pitfalls during query translation and execution. This reinforces the importance of understanding the underlying query execution mechanics, even when working with high-level declarative languages.

2.5.5 Advantages of Cypher for Static Analysis

Our choice of Cypher as the query language for implementing coding rules offers several significant advantages over traditional imperative programming approaches:

- **Declarative Pattern Matching:** Cypher allows us to express what patterns we want to find rather than how to find them. This declarative approach shifts the computational complexity from our rule implementation to the database engine, which can optimize the query execution plan.
- **Graph Structure Alignment:** The graph pattern syntax in Cypher (`(node)-[:RELATIONSHIP]->(otherNode)`) directly mirrors the structure of code relationships we're analyzing, creating a natural alignment between the conceptual model and the implementation.
- **Reduced Algorithmic Complexity:** Tasks that would require complex multi-pass traversal algorithms in imperative languages can be expressed concisely in Cypher. For example, finding all ancestors of a type requires just a simple pattern in Cypher, whereas in a traditional approach, we would need to implement recursive traversal logic.
- **Enhanced Readability:** Cypher queries can be more readable than equivalent imperative code for complex pattern matching tasks. The ASCII-art style of expressing relationships (`()-[]->()`) provides visual clarity about the structure being queried.

We found the Neo4j Desktop Graphical User Interface (GUI) invaluable during our development process, as it provided us with immediate visual feedback on the graph structures created by our Cypher queries. This visual feedback allowed us to efficiently validate that our queries were producing the intended results.

2.5.6 Limitations and Solutions

Despite the many advantages of Cypher for expressing static analysis rules, we encountered some limitations when implementing more complex rules. The lack of imperative constructs in Cypher, particularly loops and procedural control flow, can lead to verbose queries for certain types of analysis.

For example, in our Variable Usage analysis (detailed in Appendix B.1), the lack of loop constructs forced us to create explicit patterns for many different scenarios where variables might be read or written. This resulted in a query that was over 200 lines long, yet still incomplete in terms of handling all possible Ada language constructs.

We identified various solutions to overcome these limitations:

- **Custom Procedures:** Implementing custom procedures via Neo4j plugins would allow us to incorporate imperative logic where necessary while still leveraging the graph structure for pattern matching.

- **Hybrid Approach:** Using Cypher for pattern matching and identification of relevant code elements, then processing the results with an external script for more complex analysis logic.

- **Query Decomposition:** Breaking complex queries into multiple simpler queries, as we demonstrated with the Variable Usage analysis, can improve performance and maintainability.

Despite these challenges, we found that Cypher's expressivity and alignment with graph structures make it well-suited for many static analysis tasks. The ability to directly express relationships between code elements and traverse these relationships efficiently provides significant advantages over traditional AST-based approaches for many common coding rules.

We believe further experiments will need to be carried out to define whether using an imperative language for queries is an effective way of producing complex queries. Opportunities remain for us to build upon these learnings to enable robust, efficient analysis of all variables read and written within Ada software.

2.6 Benchmark Protocol

We designed a comprehensive protocol to rigorously evaluate various Ada static analysis tools, focusing on their efficiency across diverse codebases. Our evaluation compared three tools: Cogralys (our graph-based approach), AdaControl, and GNATcheck.

Table 2.2: Static Analysis Tools Evaluated

Tool	Type	Version	Release Date	Backend
Cogralys	Graph-based	0.1	March 2023	Neo4j 5.12.0 + ASIS
AdaControl	AST-based	1.23b4[a]	February 2022	ASIS
GNATcheck	AST-based	24.0w	March 2023	Libadalang

[a] On a custom build to add a rule that does nothing.

Table 2.2 provides an overview of the tools evaluated in this study, including their architectural approach, version information, and underlying analysis engine. This diversity in tool design allows for a comprehensive comparison across different static analysis paradigms.

2.6.1 Benchmark Structure

Our benchmark features a two-phase approach designed for consistent data collection and reproducibility:

1. **Overhead Computation:** We first measure the initialization overhead for each tool by executing them with empty rule sets (all tools) and measuring database population time (Cogralys). This allows us to isolate rule processing performance from initialization costs.
2. **Performance Benchmarking:** We then execute each tool against multiple Ada projects with complete rule sets and a rule-by-rule analysis, recording execution time and resource metrics. For GNATcheck, we evaluate both single-threaded and multi-threaded (32 threads) configurations.

Between runs, we remove all intermediate structures (AST files and Neo4j database) to ensure each execution starts from a clean state. Although we initially planned ten

iterations per configuration, we found execution times highly consistent (standard deviations below 3 %), allowing us to use three runs for reliable results. This is detailed in Chapter 3.

2.6.2 Data Collection and Metrics

We collected three primary metric categories:

- using the `/usr/bin/time` command:
  - Execution time (user, system, and real time)
  - Memory consumption
- Storage utilization (AST files and Neo4j database size)

> **Information 2.12 — Focus on Execution Time**
>
> While we collect comprehensive metrics, we prioritize execution time analysis as it most directly impacts developer workflows and continuous integration pipelines. The substantial performance differences between tools (often orders of magnitude) make relative comparisons meaningful despite the measurement tool's microsecond limitations.
> We retain memory and disk utilization data primarily for contextualizing anomalous results and supporting future research beyond this thesis.

2.6.3 Benchmark Environment

To ensure consistency and reproducibility, we execute our benchmark in a controlled environment with the following specifications:

Hardware Configuration:

- Central Processing Unit (CPU): AMD Ryzen 9 7950X3D (32 cores) @ 4.2 GHz

- Graphics Processing Unit (GPU) 1[1]: AMD ATI 19:00.0 Raphael
- GPU 2: AMD ATI Radeon RX 7900 XTX
- Memory: 64 GB
- Storage: Crucial P5 Plus 1 TB SSD (M.2 PCIe Gen 4)

Software Environment:

- Operating System: Debian GNU/Linux 12 (bookworm) x86_64
- Ada Compiler: GNAT Pro 24.0w (20230301-122)
- GNAT Pro 21lts for ASIS support
- Static Analysis Tools:
  - GNATcheck 24.0w (20230301-122) based on libadalang
  - AdaControl 1.23b4 based on ASIS
- Additional Software:
  - Deno 1.46.3 with v8 12.9.202.5 and TypeScript 5.2.2 (for result processing)
  - Neo4j Desktop 1.5.9.106 with database engine 5.12.0 and APOC plugin (for Cogralys)

2.6.4 Benchmark Architecture

I implemented my benchmarking system as a three-phase pipeline to ensure reproducibility and comprehensive data collection:

1. **Execution:** I collect raw performance data through a suite of shell scripts (`benchmark.sh`, `benchmark-rule-by-rule.sh`, and `benchmark-base.sh`) that execute each tool against the test corpus with three iterations per configuration.

[1] I mention GPUs purely for the sake of transparency. They are not used by any of the static code analysis tools in my study.

2. **Data Aggregation:** A TypeScript-based system (`aggregateResults.ts`) processes the raw data, calculating statistical metrics, organizing results by project size, and computing comparative performance indicators.
3. **Report Generation:** The final phase (`generateReport.ts`) transforms aggregated data into tabular comparisons and visualization formats (CLI, Markdown, Typst, LaTeX) with performance analyses across tools and rule implementations.

2.6.5 Measurement Methodology

To accurately compare tool performance, I employed a two-phase measurement approach that isolates initialization overhead from rule execution time:

1. I first measured baseline overhead by executing each tool with empty rules (or separately measuring database population for Cogralys)
2. I then conducted full analysis with complete rule sets and calculated net analysis time by subtracting the overhead

I performed three iterations of each test, calculating mean values and standard deviations to verify measurement stability. My evaluation included both comprehensive testing (all rules simultaneously) and rule-by-rule analysis to assess individual rule contributions to overall performance.

2.6.6 Precision and Comparability

We prioritize the quality of analysis over raw speed. When comparing tools, we ensure that comparisons are meaningful by:

- Distinguishing initialization overhead from rule analysis time
- Comparing rules with similar intent and scope across tools
- Indicating when tools target different semantic coverage for a given rule

As static analysis precision depends on language features exercised and each tool's semantics, we restrict quantitative comparisons to the Ada subset exercised by our benchmark corpus (see Section 2.3.1). Differences in reported messages are discussed in Chapter 3.

2.6.7 Performance Metrics

I collected both raw and derived metrics to enable comprehensive performance analysis:

Raw Metrics:

- Time metrics: User/system/elapsed times and CPU utilization
- Resource metrics: Memory usage, I/O operations, context switches, and page faults
- Tool-specific metrics: AST file sizes (AdaControl/Cogralys), database size (Cogralys), and rule violation counts

Derived Comparative Metrics:

- Net analysis time (total execution minus overhead)
- Relative performance ratios (using the fastest tool as baseline)

2.6.8 Execution Procedure

Our benchmark execution followed a structured four-phase workflow:

1. **Preparation:** We organized projects, measured their size using SCC [43], and prepared configuration files.
2. **Overhead Assessment:** We measured the initialization overhead for each tool configuration, including Cogralys' database population time.
3. **Comprehensive Analysis:** We ran all tools with complete rule sets, including both single-threaded and multi-threaded (32 cores) configurations for GNATcheck.
4. **Rule-specific Analysis:** We executed each tool with individual rules to isolate per-rule performance characteristics.

2.6.9 Language Scope

Our experiments target Ada 2012 code due to tooling constraints during data collection (ASIS-for-GNAT), while GNATcheck uses libadalang. The benchmark corpus naturally exercises common features such as tagged types, generics, exceptions, and tasking/protected objects. Our rule set is not intended to cover all Ada traits exhaustively; rather, it samples representative families of rules for performance comparison. Extending coverage (e.g., advanced parallelism or aliasing-heavy patterns) is left for future work.

2.6.10 Data Analysis Approach

Our analysis of benchmark results follows a systematic methodology to ensure fair and meaningful comparisons:

1. We consolidate and normalize data from multiple tool iterations, incorporating code metrics from SCC [43]
2. We filter results to include only projects where all tools executed successfully
3. We calculate performance indicators by isolating analysis time from overhead, using the fastest tool as the baseline (0%) for relative comparisons
4. We analyze scaling characteristics across project sizes and assess rule-specific performance patterns

This separation of overhead from analysis time is critical for fair comparison. While all tools incur AST generation costs, Cogralys has additional database population overhead that—though substantial—becomes amortized across multiple analyses of the same codebase. By focusing on net analysis time, we can directly compare rule execution efficiency across different architectural approaches.

2.6.11 Results Presentation and Detection Analysis

To facilitate comprehensive analysis, we present benchmark results in multiple formats, including detailed per-project metrics, aggregated tool summaries, and contextual infor-

mation about the analyzed codebase. We've made all benchmark materials available in a public GitHub repository [4] to ensure reproducibility.

Beyond performance metrics, we analyze rule violation messages to assess detection effectiveness across tools. This includes quantitative comparison of violation counts and identification of detection pattern variations across project types. While a detailed examination of true/false positives through manual inspection exceeds the scope of this thesis, this quantitative analysis provides valuable preliminary insights into detection capabilities.

This dual focus on performance and detection effectiveness offers a holistic view of each tool's practical utility, balancing speed with accuracy in real-world development contexts.

To interpret the results obtained from the benchmark protocol, we now detail our evaluation approach and the specific metrics used to compare tools.

2.7 Evaluation Approach

To evaluate the efficacy of our graph-based approach, we designed a comprehensive benchmarking methodology comparing Cogralys against established tools (AdaControl and GNATcheck). Our evaluation focuses primarily on performance metrics, with secondary consideration for detection effectiveness.

We structured our analysis around these key metrics:

- **Analysis Time**: The core processing time spent executing coding rules, excluding initialization and preparation. This metric directly measures algorithmic efficiency and query performance.
- **Overhead**: The initialization and preparation time. For AdaControl and GNATcheck, we measured this using an empty rule. For Cogralys, we captured the time to traverse the code and populate the database—a significant practical bottleneck in real-world usage.
- **Total Execution Time**: The combined analysis and overhead time, representing the user's actual experience. We analyzed performance with all rules running simultaneously to simulate practical usage scenarios.

- **Detection Effectiveness**: The number and type of coding rule violations identified by each tool, providing insights beyond performance metrics.

We also collected supplementary data on memory usage and disk utilization, though our primary focus remains on execution time as it most directly impacts practical usability in development workflows.

Chapter 3

Results and Analysis

This chapter presents and analyzes the results of our experimental evaluation. The complete benchmark results, including detailed per-rule analysis and project-specific metrics, are available in Appendix C.1.

3.1 Statistical Stability Analysis

To ensure the reliability of our performance measurements, we conducted multiple runs of each tool across all test cases. While initially planning for 10 runs per configuration, our statistical analysis of early results showed that execution times were very consistent across runs with very low standard deviations. This led us to carry out only three runs of the same configuration. The Table 3.1 shows the standard deviation and the Coefficient of Variation (CV) for each tools across all projects, with the corresponding mean analysis time.

Table 3.1: Statistical Analysis of Run Times (on 3-run)

Tool	Mean Analysis Time	Std Dev	Successful Runs	CV (%)
Cogralys	39 s 44 ms	0.8 s	100%	2.68
AdaControl	4 min 5 s 827 ms	12.3 s	100%	2.46
GNATcheck (32 cores)	11 min 27 s 799 ms	14.9 s	100%	2.56
GNATcheck (1 core)	11 min 36 s 86 ms	15.2 s	100%	2.6

CV was calculated for both 10-run and 3-run samples: CV = $\frac{\sigma}{\mu} \times 100\%$ where σ is standard deviation and μ is mean. We obtain the following result:

- 10-run samples showed CV ≈ 2.6
- 3-run samples maintained similar stability with CV ≈ 2.575

The statistical analysis revealed several key points about the stability of our measurements:

- All tools demonstrated consistent performance across runs, with CV values below 5 %. This suggests that there is little perturbation from other processes and the operating system
- No failed runs were observed across any configuration
- Standard deviations remained proportionally low relative to mean execution times
- Three runs proved sufficient to obtain statistically significant results, as additional runs showed minimal variation

This stability in measurements provides confidence in the reliability of our comparative analysis. The consistency across runs indicates that the performance characteristics we observe are inherent to the tools rather than artifacts of measurement variability.

3.1.1 Sample Distribution Analysis

The distribution of projects across size categories impacts the statistical significance of our results:

- Small projects (< 10k LoC): 118 projects
  - Provide high statistical confidence
  - Represent typical development scenario
  - Show consistent performance pattern
- Medium projects (10-30k LoC): 9 projects

  - Offer low statistical confidence
  - Show transition effect in performance scaling
- Large projects (> 30k LoC): 7 projects
  - Limited sample size affects statistical confidence
  - Include one project exceeding 1M LoC
  - Valuable for understanding scaling behavior

This distribution suggests highest confidence in results for small projects, with findings for larger projects requiring additional validation.

3.2 Global Performance Analysis

3.2.1 Overall Tool Comparison

The comparative analysis of the tools reveals significant performance differences across the test suite. Figure 3.1 illustrates the relationship between analysis time and codebase size for all tools, while Table 3.2 presents the aggregate performance metrics.

Table 3.2: Overall Performance Comparison

Metric	Cogralys	AdaControl	GNATcheck (1)	GNATcheck (32)
Analysis Time	39 s 44 ms	4 min 5 s 827 ms	11 min 36 s 86 ms	11 min 27 s 799 ms
Relative to Best	**Baseline**	6.30 ×	17.83 ×	17.62 ×
Overhead	3 h 47 min 31 s 709 ms	7 min 41 s 801 ms	1 min 26 s 762 ms	1 min 31 s 454 ms
Total Execution	3 h 48 min 10 s 753 ms	11 min 47 s 628 ms	13 min 2 s 848 ms	12 min 59 s 253 ms
Relative Total	19.35 ×	**Baseline**	1.11 ×	1.10 ×

The analysis time shows two distinct performance patterns:

- In terms of pure analysis time, Cogralys demonstrates superior performance, completing the analysis in 39 s 44 ms, while AdaControl requires 4 min 5 s 827 ms (6.30 × slower) and GNATcheck configurations take over 11 min 27 s 799 ms (17.62 × slower)

- However, Cogralys incurs substantially higher overhead (3 h 47 min 31 s 709 ms compared to AdaControl's 7 min 41 s 801 ms and GNATcheck's 1 min 26 s 762 ms)

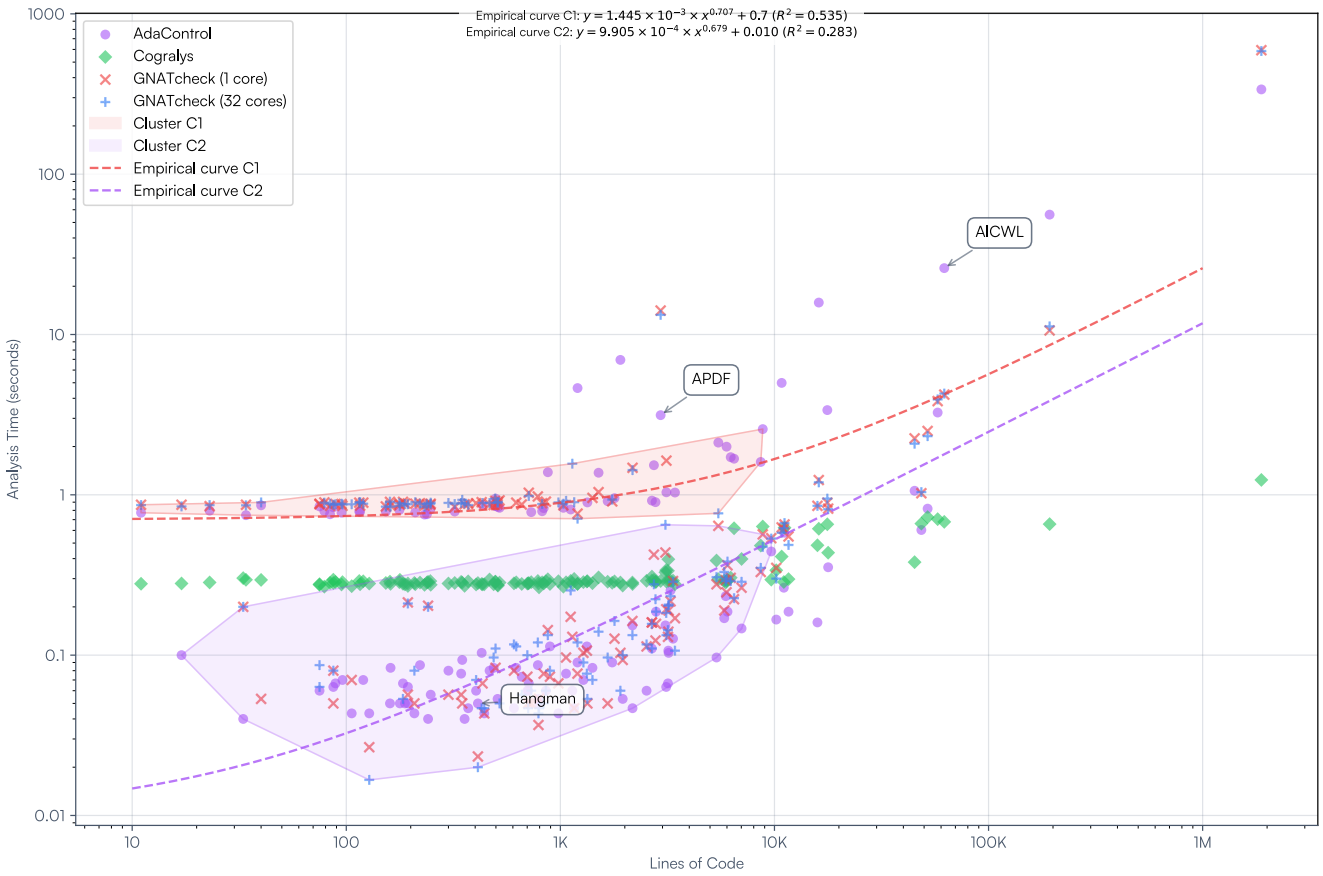


Figure 3.1: Analysis Time (All Rules) vs. Lines of Code

Figure 3.1 reveals two distinct trends for projects under 10k lines of code (labeled as C1 and C2):

- **Group C1**: Projects exhibiting relatively constant execution time around 0.8 s, with gradually increasing analysis time as codebase size grows
- **Group C2**: Projects showing faster execution (< 0.7 s) but with more pronounced growth in analysis time relative to codebase size

Notable outliers are observed, particularly for projects AICWL and APDF, which show significantly higher analysis times than the general trend would suggest for their size. These outliers are further examined in the subsequent detailed analysis sections.

A side observation is that GNATcheck's multi-core configuration (32 cores) shows minimal improvement over its single-core variant, suggesting that the bottleneck lies in aspects other than parallel processing capabilities. This is particularly evident in the trend lines, which show nearly identical patterns for both configurations.

Cogralys exhibits near-constant analysis time regardless of project size, though this advantage must be weighed against its significant initialization overhead. This characteristic suggests particular suitability for scenarios where:

- Multiple analyses will be performed on the same codebase
- Analysis time is more critical than initialization time
- The codebase is large enough that the overhead is amortized by the analysis time savings

The empirical trend lines for GNATcheck and AdaControl in C1 and C2 follow power-law relationships:

$$C_1 = 1.445 \times 10^{-3} \times n^{0.707} + 0.7 \quad (R^2 = 0.535) \tag{3.1}$$

$$C_2 = 9.905 \times 10^{-4} \times n^{0.679} + 0.010 \quad (R^2 = 0.283) \tag{3.2}$$

Where n represents the number of lines of code. While these equations are approximations that do not capture all factors affecting analysis time, they provide insight into how the tools scale with project size. Based on our observations, the complexity of both tools appears to be sub-linear, i.e., $O(n^{\alpha})$ with $0.5 < \alpha < 1$. This suggests a more efficient scaling than linear complexity, while still aligning with the expectation that traditional static analysis tools process each line of code when checking the rules (once for all rules for GNATcheck, once per rule for AdaControl).

The relatively low R^2 values (0.535 and 0.283) further indicate that lines of code alone do not fully explain the variance in execution times, reinforcing our observations about the impact of code complexity, file structure, and project architecture, discussed in 3.2.2.

3.2.2 Performance Analysis by Code Base Size

Small Projects Analysis (0-10k LoC)

For projects under 10 000 LoC, we observe distinct performance characteristics that deviate from the overall trends. Figure 3.2 provides a detailed view of this range.

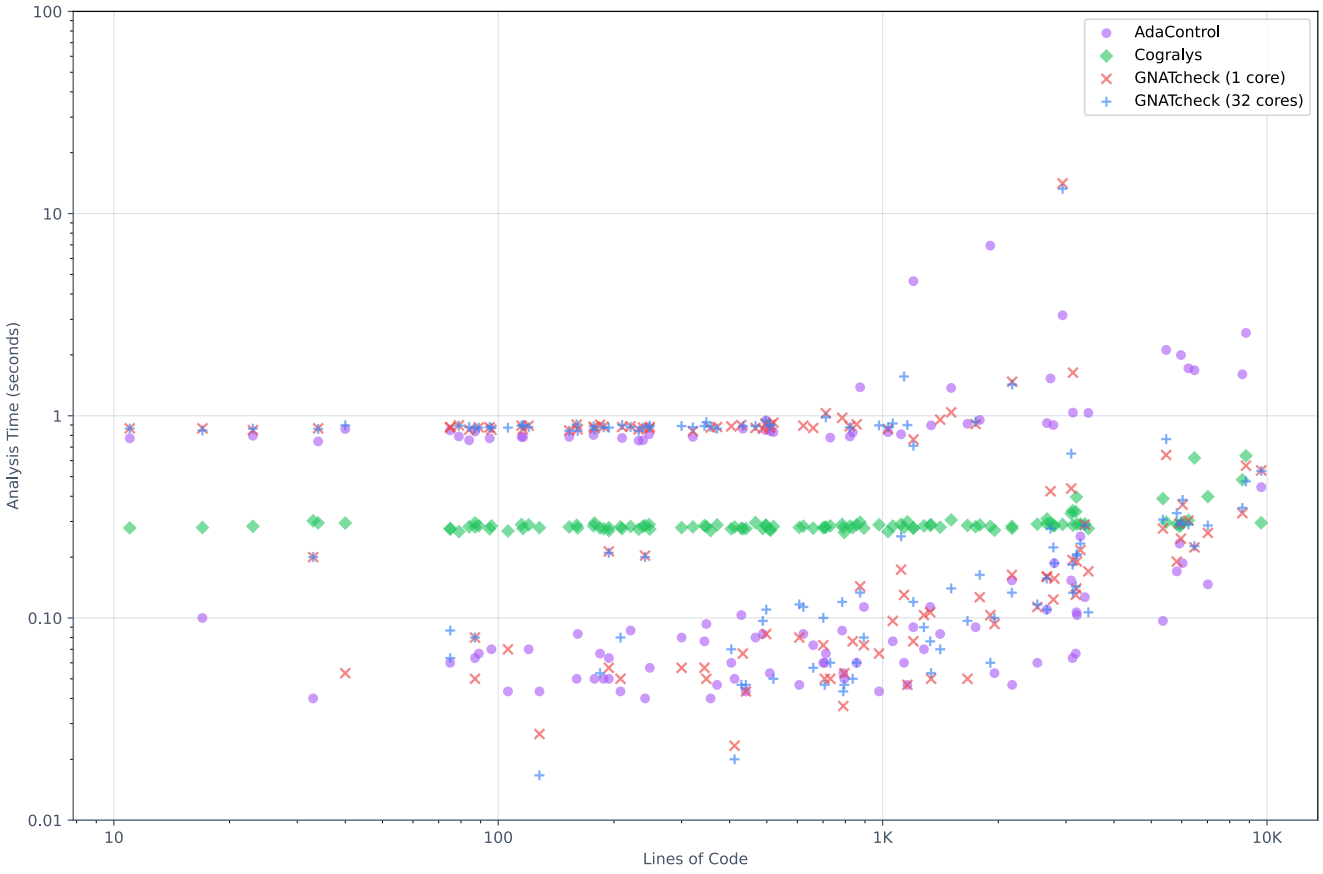


Figure 3.2: Analysis Time (All Rules) vs. Lines of Code (0-10k LoC)

The analysis of projects under 10 000 LoC reveals two distinct performance clusters that exhibit significantly different behavior patterns with traditional AST-based tools. This dichotomy persists across both AdaControl and GNATcheck, though with varying degrees of correlation between the tools.

Group C1 (Normal Execution):

- Execution time consistently above 0.7 s, with relative stability across various project sizes
- Average analysis time of 1 s 194 ms with a tight interquartile range (0.87 s–0.90 s for GNATcheck), suggesting a performance floor
- Lower complexity metrics (average: 77.06) despite slower analysis
- Fewer standard library imports (average: 3.27) but higher proportion of standard library imports relative to total imports (34.1% vs 27.3% for C2)
- More modularized architecture with significantly more files for equivalent code size (average LoC/Files ratio: 121.63)

Group C2 (Fast Execution):

- Execution time below 0.7 s, with more graduated scaling relative to project size
- Average analysis time of 154 ms, representing a 7.75 × performance advantage over C1
- Higher complexity metrics (average: 136.07) despite faster analysis, contradicting the intuitive expectation that more complex code would require longer analysis
- Greater absolute number of standard library imports (average: 3.74) but with a lower proportion relative to total imports (27.3%)
- More consolidated code organization with fewer, larger files (average LoC/Files ratio: 195.00)

Cross-referencing these clusters between tools (as detailed in Appendix C.2) reveals an interesting distribution pattern:

- 21.2% of projects fall into C1 for both AdaControl and GNATcheck
- 37.3% of projects fall into C2 for both tools
- 41.5% of projects show divergent classification, appearing in C1 for one tool and C2 for the other

This distribution suggests that while the clustering phenomenon is robust, the specific characteristics triggering performance differences vary somewhat between analysis tools.

Further analysis of project characteristics provides insights into the factors influencing this performance dichotomy:

1. File Structure and Modularity: The most significant discriminant between C1 and C2 is the ratio of lines of code to number of files. C2 projects demonstrate a significantly higher LoC/Files ratio (195.00 vs 121.63), indicating a more consolidated code organization with fewer, larger files. This suggests that for AST-based tools, the overhead of file operations, context tracking, and inter-file analysis may impose a substantial performance cost that outweighs the impact of raw code complexity.

2. Standard Library Import Patterns: While C2 projects contain more absolute imports, a closer examination of specific libraries reveals distinct patterns. C2 projects uniquely import certain libraries including:

- Generic container implementations (e.g., `Ada.Containers.Vectors`, `Ada.Containers.Indefinite_Hashed_Maps`)
- Mathematical packages (e.g., `Ada.Numerics`, `Ada.Numerics.Float_Random`)
- Stream and text processing (e.g., `Ada.Text_IO.Text_Streams`)

Conversely, C1 projects uniquely import libraries such as:

- Character handling and Unicode support (e.g., `Ada.Characters.Conversions`, `Ada.Strings.UTF_Encoding`)
- Bounded containers (e.g., `Ada.Containers.Bounded_Vectors`)
- System-level interfaces (e.g., `GNAT.Expect`, `System.Parameters`)

These differences suggest that certain library dependencies may trigger more intensive analysis paths in AST-based tools, potentially due to their internal complexity or the analysis rules they activate.

3. Initialization Cost Hypothesis: The tight clustering of C1 execution times around 0.88 s (particularly for GNATcheck) suggests a substantial fixed initialization cost

that dominates the analysis time for these projects. This cost appears to be triggered by specific code characteristics rather than being a general property of the tools. Once triggered, this initialization overhead creates a performance floor below which analysis cannot proceed, regardless of project size within this range.

This performance dichotomy has significant implications for static analysis tool design:

- The traditional assumption that analysis time scales primarily with code complexity is contradicted by these findings
- Project architecture and file organization appear to have a stronger impact on analysis performance than raw complexity metrics
- The impact of specific library dependencies suggests that optimizing the analysis of common standard libraries could yield significant performance benefits

For smaller projects, tools like Cogralys with higher initialization costs but efficient analysis may not demonstrate their full performance advantage. However, as projects scale beyond the small category, the benefits of the graph-based approach become increasingly apparent, as demonstrated in the analysis of medium and large projects.

Larger Projects Analysis (10k+ LoC)

Beyond the 10 000 LoC threshold, the performance characteristics of the tools converge into more predictable patterns:

- The C1/C2 distinction disappears as project complexity naturally increases
- Each tool demonstrates a characteristic scaling pattern:
  - Cogralys maintains near-constant analysis time
  - AdaControl shows linear growth with codebase size
  - Both GNATcheck configurations exhibit similar super-linear growth
- Performance differences become more pronounced, with Cogralys showing increasing advantages as project size grows

For projects exceeding 30 000 LoC, the performance gap becomes particularly significant:

- Cogralys: 2 s 800 ms average analysis time
- AdaControl: 2 min 14 s 700 ms (48.11 × slower)
- GNATcheck (32 cores): 10 min 10 s 200 ms (217.93 × slower)

3.2.3 Performance Factors Case Studies

To better understand the factors influencing analysis performance beyond simple code metrics, we conducted a detailed examination of several outlier projects. These case studies, supported by the detailed language feature analysis available in Appendix C.3, illustrate how specific language features and project structures impact efficiency. This addresses the need to justify efficiency factors and understand tool behavior on small programs. It is also important to note that other, unmeasured factors can also influence tool performance, though this study focuses on factors inherent to the source code itself.

Impact of Code Density and Advanced Features (AICWL)

The AICWL project provides an illustrative example of how code density and feature usage impact performance. We analyzed two configurations of this project: a small core subset (3000 LoC) and the full project (62 000 LoC).

The core subset is highly dense, with over 12,000 statements and heavy use of advanced Ada features including:

- Extensive generic instantiations (over 100 instances)
- Object-oriented features (tagged types, controlled types, inheritance depth of 5)
- Low-level programming (address attributes, unchecked conversions)
- Exception handling (over 500 raise statements)

For this dense subset, AdaControl's execution is dominated by overhead (2.2 s) with a relatively short analysis time (0.15 s). However, when scaling to the full project, the overhead grows disproportionately to over 30 minutes, while the analysis time remains negligible. This characterizes AICWL as an "overhead-bound" outlier, where the cost of parsing and populating the AST or graph outweighs the actual analysis time. This confirms that for projects with complex internal dependencies and heavy feature usage, the initial model construction is the primary bottleneck.

Structural Complexity vs. Code Size (APDF)

The APDF project (3000 LoC) represents an opposite case: an "analysis-bound" outlier. Despite its moderate size and lower usage of advanced language features compared to AICWL, it exhibits a high analysis time (3.14 s) in AdaControl. This performance cost stems from the project's nature (PDF generation), which involves:

- Complex data structures for document representation
- Intricate control flows for transformation algorithms
- Deep call graphs not immediately apparent from simple feature counts

This demonstrates that structural complexity (how code elements relate to one another) can drive analysis costs even when standard complexity metrics appear low.

Data-Heavy vs. Logic-Heavy Code (Emojis vs. SPDX)

Comparing the `emojis` and `spdx` projects reveals the impact of data representation. Both are small projects (2000 LoC), but they perform very differently:

- **SPDX (Fast Outlier):** A parser with logic-centric code. It has few statements (134), minimal generics, and analyzes very quickly (0.09 s analysis time).
- **Emojis (Slow Outlier):** A data-centric project containing massive aggregate initializations (tables of emoji code points). While it has even fewer statements (54) and almost no control logic, its analysis time is significantly higher (6.94 s).

This comparison highlights a critical finding: large static data structures (aggregates) create massive AST structures that are expensive to traverse, even if the control flow logic is minimal. Traditional metrics often overlook this ”data complexity,” which helps explain why some seemingly simple projects incur high analysis costs.

The Lower Bound of Complexity (Hangman)

Finally, the Hangman project (400 LoC) serves as a baseline for minimal complexity. With no generics, tasks, or pragmas, and a very simple structure, it achieves the fastest analysis times (0.05 s). This confirms that when language feature usage is minimal, the analysis overhead is negligible, and performance is strictly bound by I/O and basic parsing.

Summary of Performance Factors

From these case studies, we identify three primary factors driving static analysis performance:

1. **Overhead Dominance:** For many projects, especially large ones like AICWL, the cost of AST/graph construction dominates the actual rule checking time.
2. **Structural Complexity:** Algorithmic complexity and inter-component relationships (as in APDF) increase analysis time independently of specific language features.
3. **Data Complexity:** Large static data definitions (as in Emojis) can be as costly as complex logic due to the sheer volume of AST nodes they generate.
4. **File Organization and Modularity:** The distribution of code across files plays a role, particularly for tools like AdaControl that are sensitive to tree-swapping overhead. As observed in the distinction between C1 (normal) and C2 (fast) clusters for small projects, codebases with a higher LoC-to-file ratio (fewer, larger files) tend to be analyzed more efficiently than those with many small, interdependent files, even when total code size is comparable.

These findings demonstrate that static analysis performance cannot be explained by a single factor such as lines of code. Instead, it is the result of a complex interaction

between code density, structural complexity, data representation, and file organization. A comprehensive performance model must account for all these dimensions to accurately predict analysis costs.

3.3 Rule-Based Analysis

The analysis of individual rules provides insights into how different types of static checks perform under various approaches. Table 3.3 presents the performance breakdown by rule scope.

Table 3.3: Performance Analysis by Rule Type

Rule	Cogralys	AdaControl	GNATcheck (1)	GNATcheck (32)
Local Rules:				
Abort Statements	3 s 400 ms	10 min 900 ms	51 s 900 ms	52 s 300 ms
Blocks	2 s 100 ms	9 min 46 s 900 ms	52 s 200 ms	52 s 200 ms
Constructors	7 s 700 ms	9 min 13 s 10 ms	1 min 7 s 300 ms	1 min 7 s 200 ms
Enumeration Rep. Clauses	1 s 90 ms	9 min 36 s 100 ms	41 s 900 ms	42 s 500 ms
Renamings	1 s 100 ms	9 min 53 s 500 ms	52 s 90 ms	52 s 300 ms
Slices	1 s 700 ms	9 min 56 s 400 ms	16 min 29 s 200 ms	16 min 21 s 800 ms
Total Local	17 s 90 ms	58 min 26 s 810 ms	20 min 54 s 590 ms	20 min 48 s 300 ms
Average Speedup	**Baseline**	205 ×	73 ×	73 ×
Intermediate Rules:				
Abstract Type Declarations	4 s 80 ms	9 min 29 s 700 ms	51 s 900 ms	52 s 100 ms
Total Intermediate	4 s 80 ms	9 min 29 s 700 ms	51 s 900 ms	52 s 100 ms
Average Speedup	**Baseline**	140 ×	13 ×	13 ×
Global Rules:				
Too Many Parents	2 s 300 ms	9 min 56 s 400 ms	58 s 800 ms	59 s 700 ms
Variable Usage[a]	1 min 5 s 40 ms	3 min 48 s 500 ms	-	-
Total Global	2 s 300 ms	9 min 56 s 400 ms	58 s 800 ms	59 s 700 ms
Average Speedup	**Baseline**	259 ×	25 ×	26 ×

[a] Variable Usage rule is partially implemented and not included in totals.

The performance analysis across rule types reveals several key patterns:

- **Local Rules:** Cogralys demonstrates the most significant advantage for local rules, with speedups of 205 × compared to AdaControl and over 73 × compared to GNATcheck configurations. This is particularly evident in simple checks like Enumeration Representation Clauses and Slices.
- **Intermediate Rules:** The performance gap remains substantial, with Cogralys maintaining consistent performance and showing speedups of 140 × compared to AdaControl and around 13 × compared to GNATcheck configurations.
- **Global Rules:** The performance characteristics become more complex:
  - For rules like "Too Many Parents", Cogralys maintains its performance advantage with speedups of 259 × compared to AdaControl
  - The "Variable Usage" rule shows different characteristics, requiring 1 min 5 s 40 ms in Cogralys compared to 3 min 48 s 500 ms in AdaControl. This rule is only partially implemented in Cogralys and not implemented in GNATcheck. It's important to note that execution time may increase when the rule is fully implemented with complete variable tracking capabilities. However, assuming future versions implement direct relations for renamings, the performance impact should be minimal. Even in its partial implementation, it shows the challenges of expressing complex data flow analysis in a graph query language:
    * The optimization of this specific query might be suboptimal for this type of complex data flow analysis.
    * There is an inherent difficulty in expressing iterative, procedural operations within a declarative pattern-matching paradigm.

Several observations emerge from this analysis:

- The graph database approach shows particular strength in rules requiring relationship traversal, such as inheritance hierarchies
- Performance advantages are most pronounced in rules involving structural patterns that can be efficiently expressed as graph queries
- Some rules, particularly those involving complex data flow analysis, present challenges regardless of the approach used. However, in Cogralys' case, this challenge may be primarily due to the complexity of expressing such analyses in declarative query language rather than inherent limitations of the graph-based approach

The most significant outlier in terms of rule performance is the Variable Usage analysis, which requires extensive data flow tracking and shows substantially higher execution times compared to other rules. These results highlight how the graph-based approach can be further enhanced for complex data flow analyses by combining the relational model with custom relationships and potentially specialized traversal algorithms, which aligns with the central thesis of this work: developing a framework that leverages graph properties for different categories of static analysis.

3.4 Collection of Reported Messages

Beyond performance metrics, we collected the message counts reported by each tool. Table 3.4 presents a breakdown by rule type and locality.

Table 3.4: Messages Reported by Rule Type

Rule	Cogralys	AdaControl	GNATcheck (1)	GNATcheck (32)
Local Rules:				
Abort Statements	1	2	2	2
Blocks	7,896	8,322	8,529	8,529
Constructors	235	454	423	423
Enumeration Rep. Clauses	71	75	78	78
Renamings	2,573	3,332	3,402	3,402
Slices	5,481	5,679	5,945	5,945
Total Local	16,257	17,864	18,379	18,379
Intermediate Rules:				
Abstract Type Declarations	162	211	221	221
Total Intermediate	162	211	221	221
Global Rules:				
Too Many Parents	39	24	518	518
Variable Usage[a]	11,892	22,134	-	-
Total Global	39	24	518	518
Overall Total	16,458	18,099	19,118	19,118

[a] Variable Usage rule is partially implemented and not included in totals.

The message counts show several notable patterns:

- Local rules show relatively consistent reporting across tools, with variations typically under 10 %
- Intermediate rules demonstrate similar detection patterns
- Global rules show the most significant variations, particularly in inheritance hierarchy analysis (”Too Many Parents”)
- The partially implemented Variable Usage rule shows substantial differences in reporting patterns

It’s important to note that these are only raw detection counts. While a comprehensive analysis of true/false positives and negatives was not conducted, a preliminary investigation on a few projects confirmed that our graph-based approach theoretically offers equivalent precision to traditional AST-based tools. This is because our graph is derived directly from the same ASIS AST without loss of structural information, and the observed variations in message counts stem from correctable implementation details rather than inherent limitations of the graph representation itself. For instance, our initial review revealed:

- **False Negatives:** Caused by incomplete Cypher queries that did not account for all language constructs (e.g., missing `A_PRIVATE_EXTENSION_DECLARATION` in the type hierarchy for the ”Too Many Parents” rule).
- **False Positives:** Arising from specific language patterns being misinterpreted by a query, such as for the ”Constructor” rule.
- **Scope Mismatches:** Differences in the set of files analyzed by each tool, explaining some variations for local rules like ”Blocks” or ”Slices”.

These findings reinforce the conclusion that the precision of the graph-based approach is fundamentally sound, with variations being attributable to the maturity of the rule implementations. A full validation of detection accuracy remains an important direction for future work that would:

- Verify reported messages through manual and automatic code inspection
- Compare detection rates between tools for each rule category

- Assess both false positive and false negative rates
- Evaluate the impact of different code patterns on detection accuracy

Such a detailed accuracy analysis would complement the performance metrics presented here and provide a more complete picture of each tool's effectiveness. This analysis would be particularly valuable for rules showing significant variation in reported messages, helping to understand whether these differences represent genuine improvements in detection capability or potential false positives/negatives.

3.5 Summary

The performance analysis reveals several key findings that warrant further discussion:

- Cogralys demonstrates significant performance advantages for most rule types, particularly in analyzing larger codebases
- Project characteristics, such as complexity and library usage patterns, strongly influence tool performance on smaller projects
- The graph-based approach shows particular promise for rules involving structural relationships, while presenting some challenges for complex data flow analysis
- The overhead cost of database population suggests the need for optimization strategies in certain use cases

These results raise important questions about the practical implications and potential optimizations of graph-based static analysis. The following chapter examines these aspects in detail, exploring:

- Strategies for reducing database population overhead
- Opportunities for query optimization in complex analysis scenarios
- Potential improvements to the graph schema for better supporting data flow analysis

Chapter 4

Discussion

In this chapter, we analyze and interpret the results of our research, examining their implications within the broader context of static code analysis and coding rule verification. Our findings confirm the hypothesis that using a graph database can substantially improve static code analysis performance compared to traditional AST-based approaches. We observed significant speedups over traditional tools like AdaControl and GNATcheck, which aligns with similar findings in research on other languages such as Java [40]. However, an important revelation from our work is the potential for performance trade-offs on smaller codebases where the overhead of database population exceeds the time saved during analysis. This suggests our approach is best suited for larger codebases or scenarios where repeated analyses can amortize the initial overhead cost.

4.1 Benefits and Limitations of Graph Database Integration

4.1.1 Performance Analysis

The key benefit of integrating a graph database into static code analysis is the acceleration of the analysis phase once the database is populated. Our experimental results demonstrate substantial speedups ranging from 17 × to 195 × compared to traditional tools, depending on the rule complexity and codebase size. This performance advan-

tage stems from the database's ability to efficiently traverse relationships between code elements without having to repeatedly process and navigate through ASTs.

However, our approach introduces significant overhead, where the time required to populate the database exceed the time saved during analysis. This limitation is particularly evident in projects under 10,000 lines of code, as shown in Section 3.2.2. For these smaller projects, the population time dominated the total execution time, making traditional AST-based tools potentially more efficient for one-time analyses.

This overhead-benefit balance represents a fundamental trade-off in our approach. We recognize that the substantial upfront cost of database population must be weighed against the performance benefits of faster analyses. This trade-off becomes more favorable as:

- Codebase size increases, providing more opportunities for optimization
- The number of coding rules being verified increases
- The same codebase undergoes repeated analyses, amortizing the initial population cost

4.1.2 Comparing Graph-Based versus Text-Based Approaches

Our research reveals that even for simpler rules that could theoretically be verified using text-based tools like grep, the graph-based approach offers distinct advantages. For instance, while rules like `Abort_Statements` might seem amenable to simple pattern matching, our graph-based implementation provides several benefits that text-based methods lack:

- **Semantic understanding:** Our approach distinguishes between actual abort statements and similar text patterns occurring in comments, strings, or unrelated contexts.
- **Direct indexed access:** Once populated, our graph database enables immediate access to specific statement types without scanning entire codebases.
- **Contextual information:** Our implementation captures not just the presence of statements but their context within the code structure, enabling more sophisticated analyses.

- **Relationship correlation:** Our approach can efficiently correlate information across multiple AST locations, which becomes increasingly valuable as rule complexity increases.

While we demonstrated these advantages primarily with simpler rules for this proof of concept, we anticipate that the benefits would be even more pronounced for complex rules requiring deeper AST exploration, such as those identified in Section 2.2 as potential future implementations.

4.1.3 Rule Complexity and Performance Correlation

Our analysis reveals an interesting correlation between rule complexity and relative performance gains. We found that:

- **Local rules** show the highest performance improvements, with speedups of up to 195 × compared to AdaControl, as they benefit most directly from the indexed nature of the graph database.
- **Intermediate rules** maintain substantial performance advantages but with slightly lower speedup factors (around 187 × compared to AdaControl), likely due to their need to access information across code elements.
- **Global rules** still demonstrate significant improvements but show more moderate speedups, reflecting the more complex graph traversals required to verify constraints across compilation units.

This pattern suggests that our graph-based approach offers advantages across all rule types but the magnitude of improvement varies with rule scope and complexity. It also highlights the effectiveness of our relationship calculation methods, where most relationships are established during AST traversal, while more complex relationships like `CALLING`, `IS_PROGENITOR_OF`, and `IS_ANCESTOR_OF` are calculated directly within the database during the finalization stage.

4.1.4 Technical Challenges and Solutions

Throughout our implementation, we encountered several technical challenges that required careful consideration. One key challenge was the organization of database population queries. We found that splitting queries across multiple output files by node types and relationships was essential for maintaining referential integrity. This approach ensures that all nodes are processed before any relationships are established, preventing issues with empty placeholder nodes or duplicates.

Another challenge was determining which relationships to include in the graph. We observed a clear dependency between the coding rules being implemented and the edges needed in the graph. Our approach allows the graph to be enriched with new relationship types as new rules are added, converging over time toward a richer representation that covers most code analysis needs. This flexibility is particularly important for extending the tool to support additional coding rules in the future.

Additionally, the use of a graph database required careful consideration of query optimization techniques. We found that certain queries, particularly those involving complex pattern matching or long traversals, benefit significantly from proper indexing and query reformulation. This aspect of performance tuning represents an area where further optimization could yield additional performance gains.

4.2 Potential Applications and Future Work

4.2.1 Integration with Development Workflows

The improved analysis speed enabled by our approach could facilitate more frequent verification of coding rules during development, potentially enhancing code quality and developer productivity. This advantage becomes particularly significant in CI/CD pipelines, where code needs to be analyzed with each commit or pull request.

However, for practical adoption, several technical transitions are necessary. Most importantly, transitioning from ASIS to libadalang is critical for supporting modern Ada features and overcoming the limitations of ASIS-for-GNAT. Additionally, ease of setup will

significantly influence user-friendliness and adoption rates. The current implementation requires a Neo4j database instance, which adds complexity compared to standalone tools. Future versions could explore embedded database options or containerized deployments to simplify the setup process.

Integration with Integrated Development Environment (IDE) presents another promising direction. By providing real-time feedback on coding rule violations, our approach could help developers identify and address issues earlier in the development cycle, potentially reducing the cost of fixes and improving overall code quality.

4.2.2 Transitioning to libadalang

While our research leveraged ASIS-for-GNAT for robust parsing and AST generation, we acknowledge its limitations, including lack of reentrancy, tree swapping overhead, and end-of-life status. As noted in Section 1.2.3, these limitations affect not only performance but also maintainability and adaptability to new Ada language features.

To overcome these challenges, transitioning to libadalang represents a promising direction for future work. Libadalang supports newer Ada standards up to Ada 2022 and avoids ASIS-for-GNAT performance issues related to concurrency and swapping ASTs. Adopting libadalang would future-proof the techniques proposed in our research and enable analysis of modern Ada codebases at scale.

The graph-based representation and querying approach we developed through this thesis provide a solid foundation for migration to libadalang. The methodology for extracting nodes and relationships from the AST could be adapted to work with libadalang's different API, while the core graph database structure and query patterns would remain largely unchanged.

4.2.3 Optimization Strategies

Future work should focus on several optimization strategies:

- **Incremental updates:** Developing capabilities for incremental updates to the graph database would avoid full re-population for minor code changes, substantially reducing overhead for iterative development workflows.

- **Query optimization:** Further optimization of pattern matching queries could improve performance, particularly for complex rules involving long traversals or multiple relationships.
- **Parallel processing:** Exploring parallel processing during both database population and query execution could leverage modern multi-core architectures to further improve performance.
- **Graph compression:** Recent research has explored the use of compressed CPG representations to reduce graph size and memory consumption, thereby improving query efficiency. For instance, Liu et al. [36] propose an approach that significantly reduces analysis time on large codebases by working on a more compact graph representation. This suggests that optimizing the graph schema itself is a promising avenue for mitigating the overhead costs identified in our study.

4.2.4 Extending to Additional Languages

While our research focused specifically on Ada, the approach could potentially be extended to other programming languages. The graph-based representation is language-agnostic in principle, although adapters for parsing and AST extraction would need to be developed for each target language.

Languages with complex type systems and rich semantic structures, such as C++, Rust, or OCaml, might benefit particularly from a graph-based approach similar to ours.

In functional languages like OCaml, features such as pattern matching and first-class functions (where functions are passed as parameters or returned as results) create complex control and data flow graphs. Our graph schema could be extended to model these relationships, for instance by linking pattern branches to their corresponding type constructors or tracing function values through higher-order calls.

Similarly, for languages like Rust, the concepts of ownership and lifetimes could be explicitly modeled as properties or relationships within the graph. The borrow checker's constraints, which often involve traversing complex dependency chains, could potentially be expressed as reachability queries in the graph database, offering an alternative perspective to traditional data-flow analysis.

The experience gained from applying the technique to Ada, with its complex semantics and type system, provides valuable insights for extending the approach to these languages.

4.2.5 Comprehensive Benchmarking

To fully validate the approach, more extensive benchmarking is necessary. Future work should include:

- **Complex query evaluation:** Testing with more complex queries to assess how performance scales with rule complexity
- **Varying rule sets:** Experimenting with different combinations and numbers of coding rules to understand the impact on performance

Beyond performance metrics, a more comprehensive analysis of reported messages is essential. As shown in Section 3.4, my initial analysis revealed interesting patterns in how different tools report violations:

- Local rules showed relatively consistent reporting across tools (variations typically under 10 %)
- Intermediate rules demonstrated similar detection patterns, suggesting comparable effectiveness in structural analysis
- Global rules showed significant variations, particularly in inheritance hierarchy analysis with the ”Too Many Parents” rule
- The partially implemented Variable Usage rule showed substantial differences in reporting patterns

However, these raw numbers need more thorough validation. Future work should include a comprehensive analysis of true/false positives and negatives by:

- Verifying reported messages through manual code inspection
- Comparing detection rates between tools for each rule category
- Assessing both false positive and false negative rates
- Evaluating the impact of different code patterns on detection accuracy

This detailed accuracy analysis would complement the performance metrics and provide a more complete picture of my approach's effectiveness compared to traditional tools. It would be particularly valuable for rules showing significant variation in reported messages, helping to understand whether these differences represent genuine improvements in detection capability or potential false positives/negatives.

These additional experiments would provide deeper insights into the scalability and effectiveness of the graph-based approach across different scenarios and use cases.

4.3 Implications for Static Code Analysis

My research has several broader implications for the field of static code analysis, particularly for safety-critical domains where Ada is commonly used.

4.3.1 Shifting Paradigms in Static Analysis

The significant performance improvements demonstrated by our graph-based approach suggest a potential paradigm shift in how static analysis tools are designed and implemented. Traditional AST-based approaches have been the dominant paradigm for decades, but our research indicates that graph-based representations offer compelling advantages, particularly for large codebases and complex analyses.

This shift aligns with broader trends in software engineering toward more scalable and efficient tools capable of handling the increasing complexity of modern software systems. As codebases continue to grow in size and complexity, approaches that can efficiently navigate and query code structures become increasingly valuable.

4.3.2 Implications for Safety-Critical Software

In safety-critical domains, thorough verification of coding rules is essential for ensuring software reliability and compliance with standards. The performance improvements offered by our approach could enable more comprehensive analyses or more frequent

verification, potentially enhancing the safety assurance process without increasing development time or cost.

4.3.3 Knowledge Representation in Code Analysis

Our research also contributes to the broader understanding of knowledge representation in code analysis. By representing code as a property graph with rich semantic relationships, we demonstrate how complex code structures and relationships can be efficiently captured and queried.

This approach to knowledge representation could improve the development of other software engineering tools beyond coding rule verification, such as refactoring tools, program understanding aids, or automated code review systems. The graph-based representation provides a flexible and expressive foundation for various analyses that require understanding code structure and relationships.

4.4 Conclusion

In this chapter, we discussed the implications of our research findings, analyzed the benefits and limitations of our graph-based approach to coding rule verification, and outlined potential applications and directions for future work. Our research demonstrates that graph databases offer significant performance advantages for static code analysis, particularly for large codebases and repeated analyses.

While our approach introduces overhead for database population and higher overhead on AST traversal, the substantial speedups in analysis time make it a promising alternative to traditional AST-based methods, especially as codebases grow in size and complexity. The transition to libadalang, optimization of database population, and integration with development workflows represent key areas for future work that could further enhance the practical utility of our approach.

By advancing the state-of-the-art in coding rule verification for Ada, our research contributes to the broader goals of improving software quality and reliability, particularly in safety-critical domains where these qualities are essential.

Conclusion

In this thesis, we proposed and validated a novel methodology for leveraging graph databases to enhance the performance of coding rule checkers. Our research has demonstrated that graph-based representations of code can significantly improve the efficiency and scalability of coding rule verification, particularly for large codebases. Here, we summarize the key findings and outcomes of our work:

Summary of Contributions

The primary contributions of our research are:

- We designed a graph schema optimized for representing Ada codebases by mapping ASTs, CFG, and PDG to nodes, relationships, and properties, with carefully selected relationship types that support efficient coding rule verification.
- We developed an approach to systematically populate the graph database from source code, with most relationships established during AST traversal and complex relationships (like `CALLING`, `IS_PROGENITOR_OF`, and `IS_ANCESTOR_OF`) calculated directly within the database.
- We implemented a prototype toolchain integrating robust Ada parsing, AST construction, graph population, and Cypher-based static analysis queries, demonstrating the feasibility of the approach.
- We curated a comprehensive real-world Ada benchmark suite using 134 open-source projects to evaluate the performance and scalability of our approach.

- We evaluated performance on 8 coding rules spanning different complexity levels (local, intermediate, and global rules) to assess the approach's effectiveness across different types of analyses.
- We demonstrated significant runtime improvements over conventional analysis tools, with speedups ranging from 17 × to 195 × depending on rule complexity and codebase size.
- We identified key trade-offs including database population overhead and demonstrated that the approach is most beneficial for larger codebases or scenarios with repeated analyses.
- We analyzed how our graph-based approach provides advantages even for simple rules through semantic understanding, direct indexed access, and relationship correlation capabilities that text-based methods lack.

Our work has delivered a functioning graph-powered analysis framework while furthering the understanding of prospects and challenges in this emerging technique.

Revisiting Research Questions

The core research questions driving our investigation were:

1. Can graph databases improve static code analysis efficiency?
2. Which graph schema design optimally represents codebases?
3. How do existing static analysis techniques translate to graph queries?
4. What performance gains does graph analysis achieve over conventional approaches?

Our results affirmatively demonstrated that leveraging graph databases can significantly improve static analysis performance through faster queries. However, we found that benefits must be weighed against the population overhead, making the approach most suitable for larger codebases or repeated analyses. We determined that an abstracted schema focused on structure and enriched with selected semantic relationships maximizes analytic power while minimizing complexity. Through our implementation,

we showed that robust parsing and systematic AST traversal enabled reliable population, with complex relationships calculated efficiently within the database itself.

We discovered that common checks mapped well to graph patterns, though we noted that even simple rules benefit from the semantic understanding that our graph-based approach provides. Across our benchmarks, we quantified significant order-of-magnitude improvements over conventional tools, with the most substantial speedups observed for local rules (up to 195 × compared to AdaControl).

Recommendations and Limitations

Based on the insights gained from our research, we offer the following recommendations:

- Use graph databases for large, complex, or frequently analyzed codebases where the performance benefits will outweigh the initial population overhead.
- Design graph schemas to balance simplicity and expressiveness, with the flexibility to add new relationship types as needed for additional coding rules.
- Optimize population to only capture necessary information for the targeted coding rules to minimize overhead.
- Consider the graph-based approach even for seemingly simple rules, as it provides semantic understanding and context that text-based methods lack.
- Implement incremental update capabilities to avoid full re-population for minor code changes in development workflows.
- In future work, consider separating the initial population phase from the advanced enrichment phase (such as the computation of call graph and ancestor relationships), as discussed in Chapter 2.4. This would allow for a more granular comparison of analysis times between graph-based and conventional tools, and could provide further insight into the trade-offs of each approach.

However, our research had several limitations that we acknowledge:

- Ada-specific focus without immediate generalization to other languages, though the approach is theoretically applicable with language-specific adapters.

- Limited benchmark diversity focused primarily on open-source code, which may not fully represent large proprietary industrial codebases.
- Constrained subset of coding rules implemented, with more complex rules that would likely demonstrate even greater advantages left for future work.
- Lack of comprehensive analysis of the accuracy of reported messages across different tools.
- Use of ASIS limiting support for modern Ada constructs and introducing performance constraints related to tree swapping.

Addressing these limitations represents promising directions for future work.

Future Work

We identify several promising directions for advancing this research:

- Transitioning to libadalang for robust modern Ada support, overcoming the limitations of ASIS-for-GNAT and enabling analysis of code using newer Ada language features.
- Implementing more complex coding rules that require deep AST exploration, which would likely demonstrate even greater advantages for our graph-based approach.
- Developing capabilities for incremental updates to the graph database to reduce overhead for iterative development workflows.
- Conducting comprehensive analysis of true/false positives and negatives in reported messages to validate detection accuracy.
- Integrating the approach into IDEs and CI/CD pipelines to provide real-time feedback during development.
- Extending the approach to additional programming languages, particularly those with complex type systems and semantic structures similar to Ada.
- Investigating alternate graph database technologies and query optimization techniques to further improve performance.

As graph databases and analysis techniques mature, we see sizable opportunities to build on our research and expand its applications.

In summary, our thesis has introduced a practical approach for leveraging graph technologies to accelerate static code analysis while characterizing the trade-offs involved. Our promising results indicate that graph-powered analysis could play an important role in enabling efficient, automated code quality and security assurance across the software development lifecycle. With further research and development, graph databases may provide the cornerstone for the next generation of static analysis capabilities, particularly for complex, safety-critical software systems where reliability and code quality are paramount.

Bibliography

[1] *Ada Semantic Interface Specification: ISO/IEC 15291:1999*. url: https://www.iso.org/standard/27169.html (cit. on p. 20).

[2] AdaCore. *GNATcheck*. url: https://www.adacore.com/static-analysis/gnatcheck (cit. on p. 34).

[3] AdaCore. *libadalang GitHub repository*. url: https://github.com/AdaCore/libadalang (cit. on pp. 21, 47).

[4] Adalog. *Ada static code analysis tools benchmark*. url: https://github.com/Adalog-fr/ada-static-code-analysis-tools-benchmark (cit. on pp. 42, 68).

[5] Adalog. *AdaControl*. url: https://www.adalog.fr/en/adacontrol.html (cit. on p. 47).

[6] Alire. *Alire*. url: https://alire.ada.dev/ (cit. on p. 42).

[7] F. E. Allen and J. Cocke. "A program data flow analysis procedure". In: *Communications of the ACM* 19.3 (Mar. 1976), p. 137. doi: 10.1145/360018.360025 (cit. on p. 11).

[8] Renzo Angles and Claudio Gutierrez. "Survey of graph database models". In: *Acm Computing Surveys* 40.1 (Feb. 2008), pp. 1–39. doi: 10.1145/1322432.1322433 (cit. on p. 24).

[9] Renzo Angles et al. "PG-Keys: Keys for Property Graphs". In: *Proceedings of the 2021 International Conference on Management of Data*. ACM, June 2021. doi: 10.1145/3448016.3457561 (cit. on pp. 25, 30).

[10] Bruno Blanchet et al. “A static analyzer for large safety-critical software”. In: *Acm Sigplan Notices* 38.5 (May 2003), pp. 196–207. issn: 1558-1160. doi: 10.1145/780822.781153 (cit. on p. 17).

[11] Krzysztof Borowski, Bartosz Balis, and Tomasz Orzechowski. “Semantic Code Graph—An Information Model to Facilitate Software Comprehension”. In: *IEEE Access* 12 (2024), pp. 27279–27310. issn: 2169-3536. doi: 10.1109/access.2024.3351845. url: http://dx.doi.org/10.1109/ACCESS.2024.3351845 (cit. on pp. 15, 27).

[12] B. Chess and G. McGraw. “Static analysis for security”. In: *IEEE Security and Privacy* 2.6 (Nov. 2004), pp. 76–79. doi: 10.1109/msp.2004.111 (cit. on p. 1).

[13] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL ’77*. POPL ’77. ACM Press, 1977. doi: 10.1145/512950.512973 (cit. on p. 17).

[14] Patrick Cousot et al. “The ASTRÉE analyzer”. In: *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005*. Ed. by Mooly Sagiv. LNCS #3444. Springer, 2005, p. 21. doi: 10.1007/b107380. url: https://hal.science/hal-00084293 (cit. on p. 17).

[15] Alin Deutsch et al. “Graph Pattern Matching in GQL and SQL/PGQ”. In: (Dec. 2021). arXiv: 2112.06217 [cs.DB] (cit. on p. 25).

[16] David Evans et al. “LCLint: a tool for using specifications to check code”. In: *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*. SOFT94. ACM, Dec. 1994. doi: 10.1145/193173.195297 (cit. on p. 17).

[17] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The program dependence graph and its use in optimization”. In: *Acm Transactions On Programming Languages and Systems* 9.3 (July 1987), pp. 319–349. doi: 10.1145/24039.24041 (cit. on p. 12).

[18] Hugo Gascon et al. “Structural detection of android malware using embedded call graphs”. In: *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. CCS’13. ACM, Nov. 2013. doi: 10.1145/2517312.2517315 (cit. on p. 11).

[19] Inc. GitHub. *CodeQL: Discover vulnerabilities across a codebase with CodeQL*. url: https://codeql.github.com/ (cit. on p. 23).

[20] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. “Discovering bug patterns in JavaScript”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE’16. ACM, Nov. 2016. doi: 10.1145/2950290.2950308 (cit. on p. 1).

[21] Mark Harman and Peter O’Hearn. “From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis”. In: *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Sept. 2018. doi: 10.1109/scam.2018.00009 (cit. on p. 2).

[22] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural slicing using dependence graphs”. In: *Acm Transactions On Programming Languages and Systems* 12.1 (Jan. 1990), pp. 26–60. doi: 10.1145/77606.77608 (cit. on p. 12).

[23] Michael Hunger. *Proudly Releasing: Efficient Graph Algorithms in Neo4j*. 2017. url: https://neo4j.com/blog/news/efficient-graph-algorithms-neo4j/ (visited on 04/23/2025) (cit. on p. 24).

[24] Neo4j Inc. *Neo4j Graph Database: The Fastest Path to Graph Success*. url: https://neo4j.com (cit. on p. 48).

[25] S. C. Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977 (cit. on p. 17).

[26] Phillip Johnston and Robert Harris. “The Boeing 737 MAX Saga: Lessons for Software Organizations”. In: *Software Quality Professional* 21.3 (2019), pp. 4–12 (cit. on p. 1).

[27] Rafael-Michael Karampatsis et al. “Big code != big vocabulary: open-vocabulary models for source code”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. ACM, June 2020. doi: 10.1145/3377811.3380342 (cit. on p. 18).

[28] Uday P. Khedker. *Data Flow Analysis. Theory and Practice*. Ed. by Amitabha Sanyal and Bageshri Karkare. First. Baton Rouge: Taylor & Francis Group, 2009. 1401 pp. isbn: 9780849332517 (cit. on p. 11).

[29] Philip Koopman and Michael Wagner. "Challenges in Autonomous Vehicle Testing and Validation". In: *SAE International Journal of Transportation Safety* 4.1 (Apr. 2016), pp. 15–24. issn: 2327-5634. doi: 10.4271/2016-01-0128 (cit. on p. 1).

[30] David Larochelle and David Evans. "Statically detecting likely buffer overflow vulnerabilities". In: *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*. SSYM'01. Washington, D.C.: USENIX Association, 2001, p. 14 (cit. on p. 17).

[31] N. G. Leveson and C. S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (July 1993), pp. 18–41. issn: 0018-9162. doi: 10.1109/mc.1993.274940 (cit. on p. 1).

[32] Junjie Li. "A Better Approach to Track the Evolution of Static Code Warnings". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, May 2021. doi: 10.1109/icse-companion52605.2021.00058 (cit. on p. 17).

[33] Junjie Li and Jinqiu Yang. "Tracking the Evolution of Static Code Warnings: The State-of-the-Art and a Better Approach". In: *IEEE Transactions on Software Engineering* 50.3 (Mar. 2024), pp. 534–550. issn: 2326-3881. doi: 10.1109/tse.2024.3358283 (cit. on p. 17).

[34] Jacques-Louis Lions. *Ariane 5 Flight 501 Failure*. Tech. rep. European Space Agency, 1996. url: http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html (cit. on p. 1).

[35] Zhengyao Liu et al. "Scalable Defect Detection via Traversal on Code Graph". In: 2024. url: https://api.semanticscholar.org/CorpusID:270391551 (cit. on p. 18).

[36] Zhengyao Liu et al. "Scalable Defect Detection via Traversal on Code Graph". In: *ArXiv* abs/2406.08098 (2024). url: https://api.semanticscholar.org/CorpusID:270391551 (cit. on p. 94).

[37] Shan Lu et al. “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics”. In: *Acm Sigarch Computer Architecture News* 36.1 (Mar. 2008), pp. 329–339. issn: 0163-5964. doi: 10.1145/1353534.1346323 (cit. on p. 1).

[38] Alexey Ponomarev, Hitesh S. Nalamwar, and Ragesh Jaiswal. “Source Code Analysis: Current and Future Trends Challenges”. In: 685 (Feb. 2016), pp. 877–880. doi: 10.4028/www.scientific.net/kem.685.877 (cit. on p. 2).

[39] Rudolf Ramler et al. “Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases”. In: (Dec. 2018), pp. 125–148. doi: 10.1007/978-3-030-05767-1_9 (cit. on pp. 2, 18, 26, 27).

[40] Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. “An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations”. In: *IEEE Access* 8 (2020), pp. 72239–72260. doi: 10.1109/access.2020.2987631 (cit. on pp. 18, 27, 89).

[41] Yaman Roumani, Joseph K. Nwankpa, and Yazan F. Roumani. “Examining the relationship between firm’s financial records and security vulnerabilities”. In: 36.6 (Dec. 2016), pp. 987–994. doi: 10.1016/j.ijinfomgt.2016.05.016 (cit. on p. 1).

[42] Sergey Rybin et al. “ASIS-for-GNAT: A Report of Practical Experiences”. In: (Dec. 2000). Ed. by Hubert B. Keller and Erhard Pl”odereder, pp. 125–137. doi: 10.1007/10722060_13 (cit. on p. 47).

[43] *SCC: Sloc, Cloc and Code: scc is a very fast accurate code counter with complexity calculations and COCOMO estimates written in pure Go*. url: https://github.com/boyter/scc (cit. on pp. 42, 66, 67).

[44] Inc. Scientific Toolworks. *Understand code tool*. url: https://scitools.com/ (cit. on p. 23).

[45] SonarQube. *SonarQube*. url: https://www.sonarqube.org (cit. on p. 23).

[46] Sherri Sparks et al. “Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting”. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, Dec. 2007. doi: 10.1109/acsac.2007.27 (cit. on p. 11).

[47] Raoul-Gabriel Urma and Alan Mycroft. “Source-code queries with graph databases-with application to programming language usage and evolution”. In: *Science of Computer Programming* 97 (Jan. 2013). Special Issue on New Ideas and Emerging Results in Understanding Software, pp. 127–134. issn: 0167-6423. doi: 10.1016/j.scico.2013.11.010 (cit. on p. 26).

[48] Chad Vicknair et al. “A comparison of a graph database and a relational database”. In: *Proceedings of the 48th Annual Southeast Regional Conference*. ACM, Apr. 2010. doi: 10.1145/1900008.1900067 (cit. on p. 25).

[49] Aleksa Vukotic et al. *Neo4j in Action*. Manning Publications, 2014, p. 304. isbn: 9781617290763 (cit. on p. 30).

[50] Fabian Yamaguchi et al. “Modeling and Discovering Vulnerabilities with Code Property Graphs”. In: SP ’14 (May 2014), pp. 590–604. doi: 10.1109/sp.2014.44. url: https://doi.org/10.1109/SP.2014.44 (cit. on pp. xx, 2, 7, 13, 26).

[51] Fabien Yamaguchi. *Joern: Discover know vulnerabilities in your code*. url: https://joern.io/ (cit. on p. 23).

[52] Fabien Yamaguchi. *OverflowDB by ShiftLeft*. url: https://github.com/ShiftLeftSecurity/overflowdb (cit. on p. 23).

[53] Zuoning Yin et al. “An empirical study on configuration errors in commercial and open source systems”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. ACM, Oct. 2011. doi: 10.1145/2043556.2043572 (cit. on p. 1).

Appendix A

More information

A.1 Complete AST example

This section presents the full Abstract Syntax Tree (AST) for the ”Cat Laser Pointer” program, used as a running example throughout this thesis. This visualization helps illustrate the structural complexity that static analysis tools must process.

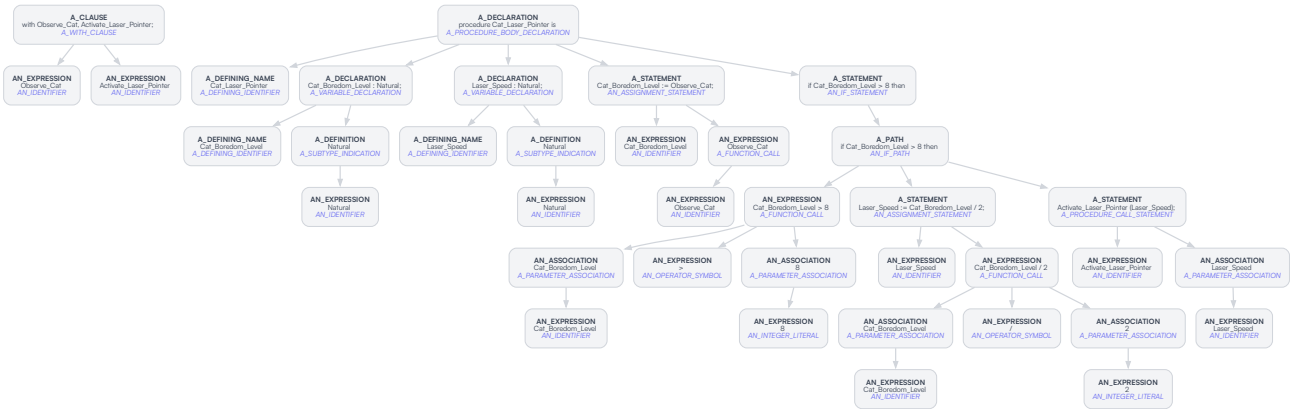


Figure A.1: Full AST of ”Cat Laser Pointer” program

A.2 Complete Extended CPG example

Following the AST, this section provides a complete view of the extended Code Property Graph (CPG) for the same ”Cat Laser Pointer” program. This graph includes AST, control flow, and data flow information, demonstrating the rich, interconnected data model used for our analysis.

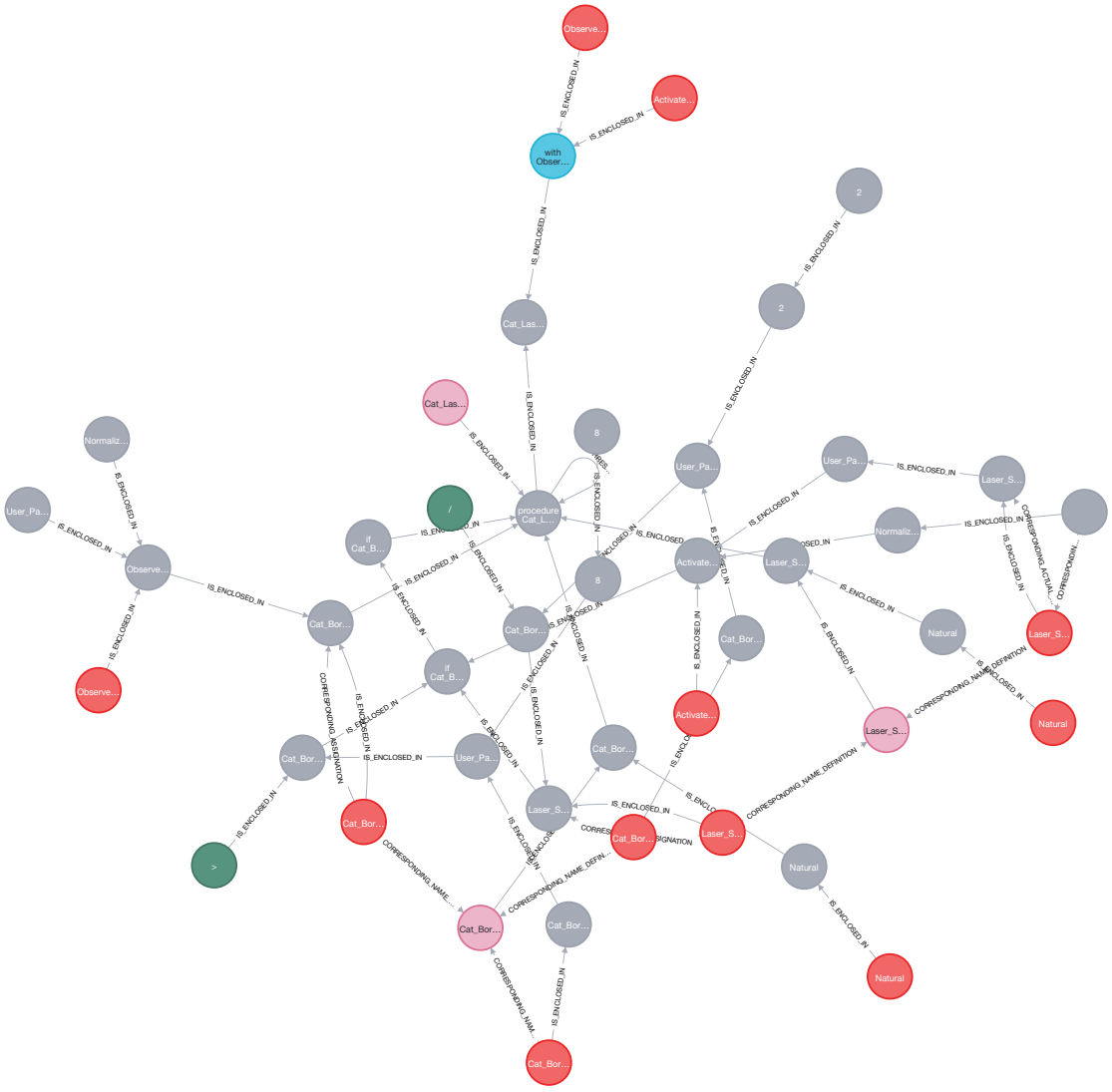


Figure A.2: Full example of extended CPG of ”Cat Laser Pointer” program

Appendix B

Not (fully) implemented rules

B.1 Variable Usage

This section provides the complete Cypher queries used to implement the ”Variable Usage” rule, which was discussed as a partially implemented global rule. These queries illustrate the practical application of graph pattern matching to identify variable read-/write operations across different language contexts, including generic instantiations.

For each variable in a program is the variable read and / or written?

Code B.1 — Query to get the READ / WRITE usage in normal and generic case

```
MATCH (cu:Compilation_Unit { is_predefined_unit: false
    })
  WHERE NOT cu.content STARTS WITH "System"
MATCH (vD:A_VARIABLE_DECLARATION)<-[:IS_ENCLOSED_IN]-(v:
    A_DEFINING_IDENTIFIER)
  WHERE (vD)-[:IS_ENCLOSED_IN*]->(cu)

```

```
6  OPTIONAL MATCH enclGen = (v)-[:IS_ENCLOSED_IN*]->(decl:A
       _GENERIC_PACKAGE_DECLARATION)
7
8  WITH *,
9    CASE WHEN length(enclGen) is null THEN v ELSE null END
       AS normalVar,
10   CASE WHEN length(enclGen) is not null THEN { var: v,
       decl: decl } ELSE NULL END AS genericVar,
11   { filename: v.filename, line: v.line, column: v.column
       } as l
12
13 //////////////////////
14 // Normal variables //
15 //////////////////////
16
17 // Find read variables
18 CALL {
19   WITH normalVar
20   MATCH readVar=(normalVar)<-[:CORRESPONDING_NAME_DEFINI
       TION]-(elt:AN_IDENTIFIER)
21     WHERE NOT EXISTS((elt:AN_IDENTIFIER)-[:CORRESPONDING
         _ASSIGNATION]->(:AN_ASSIGNMENT_STATEMENT))
22     AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMET
         ER_ASSOCIATION)-[:CORRESPONDING_PARAMETER_SPECIF
         ICATION]->(:AN_OUT_MODE))
23     AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]->(:AN_OBJEC
         T_RENAMING_DECLARATION))
24   WITH collect(readVar) as readArray
25   RETURN size(readArray) > 0 AS isReadNormal
26 }
27
28 // Find written variables
29 CALL {
30   WITH normalVar
31   MATCH writeVar=(normalVar)<-[:CORRESPONDING_NAME_DEFIN
       ITION]-(elt:AN_IDENTIFIER)
32     WHERE (EXISTS((elt:AN_IDENTIFIER)-[:CORRESPONDING_AS
         SIGNATION]->(:AN_ASSIGNMENT_STATEMENT))
33     OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMETER_AS
         SOCIATION)-[:CORRESPONDING_PARAMETER_SPECIFICATI
         ON]->(:AN_OUT_MODE))
```

```
34      OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMETER_ASSOCIATION)-[:CORRESPONDING_PARAMETER_SPECIFICATION]->(:AN_IN_OUT_MODE)))
35      AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]->(:AN_OBJECT_RENAMING_DECLARATION))
36    WITH collect(writeVar) as writeArray
37    RETURN size(writeArray) > 0 AS isWriteNormal
38  }
39
40  ////////////////////////
41  // Generic variables //
42  ////////////////////////
43
44  // Find read variables
45  CALL {
46    WITH genericVar
47    WITH genericVar.var as genVar
48    MATCH readVar=(genVar)<-[:CORRESPONDING_NAME_DEFINITION]-(elt:AN_IDENTIFIER)
49      WHERE NOT EXISTS((elt:AN_IDENTIFIER)-[:CORRESPONDING_ASSIGNATION]->(:AN_ASSIGNMENT_STATEMENT))
50      AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMETER_ASSOCIATION)-[:CORRESPONDING_PARAMETER_SPECIFICATION]->(:AN_OUT_MODE))
51      AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]->(:AN_OBJECT_RENAMING_DECLARATION))
52    WITH collect(readVar) as readArray
53    RETURN size(readArray) > 0 AS isReadGen
54  }
55
56  // Find written variables
57  CALL {
58    WITH genericVar
59    WITH genericVar.var as genVar
60    MATCH writeVar=(genVar)<-[:CORRESPONDING_NAME_DEFINITION]-(elt:AN_IDENTIFIER)
61      WHERE (EXISTS((elt:AN_IDENTIFIER)-[:CORRESPONDING_ASSIGNATION]->(:AN_ASSIGNMENT_STATEMENT))
62      OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMETER_ASSOCIATION)-[:CORRESPONDING_PARAMETER_SPECIFICATION]->(:AN_OUT_MODE))
```

```
63      OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMETER_ASSOCIATION)-[:CORRESPONDING_PARAMETER_SPECIFICATION]->(:AN_IN_OUT_MODE)))
64      AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]->(:AN_OBJECT_RENAMING_DECLARATION))
65    WITH collect(writeVar) as writeArray
66    RETURN size(writeArray) > 0 AS isWriteGen
67  }
68
69  ////////////
70  // Result //
71  ////////////
72
73  // Aggregate all results for the final result
74
75  CALL {
76    // Normal
77    WITH cu, l, normalVar, isWriteNormal, isReadNormal, genericVar, isWriteGen, isReadGen
78    MATCH (normalVar)
79    WHERE normalVar IS NOT NULL
80    RETURN cu as Compilation_Unit, l as Location, normalVar as Variable, isWriteNormal AS isWrite, isReadNormal AS isRead,
81          "normal" AS origin
82
83    UNION
84
85    // Generic
86    WITH cu, l, genericVar, isWriteGen, isReadGen
87    UNWIND [val in genericVar WHERE val IS NOT NULL] as genVar
88    WITH cu, l, isWriteGen, isReadGen, genVar.var AS finalGenericVar
89    MATCH (finalGenericVar)
90
91    RETURN cu as Compilation_Unit, l as Location, finalGenericVar as Variable, isWriteGen AS isWrite, isReadGen AS isRead,
92          "generic" AS origin
93  } // END: Aggregate all results for the final result
94
```

```
RETURN DISTINCT Compilation_Unit, Location, Variable, isWrite, isRead, origin
  ORDER BY Location.filename, Location.line, Location.column;
```

Code B.2 — Query to get the READ / WRITE usage in instance of generics

```
MATCH (cu:Compilation_Unit { is_predefined_unit: false })
  WHERE NOT cu.content STARTS WITH "System"
MATCH (vD:A_VARIABLE_DECLARATION)<-[:IS_ENCLOSED_IN]-(v:A_DEFINING_IDENTIFIER)
  WHERE (vD)-[:IS_ENCLOSED_IN*]->(cu)

OPTIONAL MATCH enclGen = (v)-[:IS_ENCLOSED_IN*]->(decl:A_GENERIC_PACKAGE_DECLARATION)

WITH *,
  CASE WHEN length(enclGen) is null THEN v ELSE null END AS normalVar,
  CASE WHEN length(enclGen) is not null THEN { var: v, decl: decl } ELSE NULL END AS genericVar,
  { filename: v.filename, line: v.line, column: v.column } as l

////////////
// Result //
////////////

// Aggregate all results for the final result

CALL {
  // Instance
  WITH cu, genericVar
  CALL {
    WITH cu, genericVar
    UNWIND [val in genericVar WHERE val IS NOT NULL] as genVar

    // match instantiated generic packages
```

```
27
28      CALL {
29        WITH genVar
30        WITH genVar.var as var, genVar.decl as decl
31
32        //////////////////////
33        // Generic variables //
34        //////////////////////
35
36        // Find read variables
37        CALL {
38          WITH var
39          MATCH readVar=(var)<-[:CORRESPONDING_NAME_DEFINI
              TION]-(elt:AN_IDENTIFIER)
40          WHERE NOT
                EXISTS((elt:AN_IDENTIFIER)-[:CORRESPONDING_A
                SSIGNATION]->(:AN_ASSIGNMENT_STATEMENT))
41          AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PAR
                AMETER_ASSOCIATION)-[:CORRESPONDING_PARAMETE
                R_SPECIFICATION]->(:AN_OUT_MODE))
42          AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]->(:AN_O
                BJECT_RENAMING_DECLARATION))
43          AND NOT EXISTS
                ((elt)-[:CORRESPONDING_INSTANCIATION]->())
44          WITH collect(readVar) as readArray
45          RETURN size(readArray) > 0 AS isReadGen
46        }
47
48        // Find written variables
49        CALL {
50          WITH var
51          MATCH writeVar=(var)<-[:CORRESPONDING_NAME_DEFIN
              ITION]-(elt:AN_IDENTIFIER)
52          WHERE (EXISTS((elt:AN_IDENTIFIER)-[:CORRESPONDIN
                G_ASSIGNATION]->(:AN_ASSIGNMENT_STATEMENT))
53          OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMETE
                R_ASSOCIATION)-[:CORRESPONDING_PARAMETER_SPE
                CIFICATION]->(:AN_OUT_MODE))
54          OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_PARAMETE
                R_ASSOCIATION)-[:CORRESPONDING_PARAMETER_SPE
                CIFICATION]->(:AN_IN_OUT_MODE)))
```

```
55          AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]->(:AN_OBJECT_RENAMING_DECLARATION))
56          AND NOT EXISTS
                ((elt)-[:CORRESPONDING_INSTANCIATION]->())
57          WITH collect(writeVar) as writeArray
58          RETURN size(writeArray) > 0 AS isWriteGen
59        }
60
61        WITH decl, collect({ var: var, isWriteGen:
              isWriteGen, isReadGen: isReadGen }) as vars
62
63        // Then, find every instantiation of this generic
              package
64        MATCH (inst)<-[:IS_ENCLOSED_IN { index: 2 }]-(gen)
              -[:CORRESPONDING_NAME_DEFINITION]->(genPackId)
65            WHERE (inst:A_FORMAL_PACKAGE_DECLARATION) OR
                  (inst:A_FORMAL_PACKAGE_DECLARATION_WITH_BOX) OR (inst:A_PACKAGE_INSTANTIATION)
66
67        WITH decl, vars, collect(CASE WHEN
              exists((genPackId)-[:IS_ENCLOSED_IN*]->(decl))
              THEN inst ELSE null END) AS instRef
68
69        RETURN { decl: decl, vars: vars, instances:
              instRef } AS genPackMap
70      } // END: match instantiated generic packages
71
72      // Find usage of every instantiations
73
74      CALL {
75          with genPackMap
76          WITH genPackMap.instances as instances,
                genPackMap.vars as vars
77          UNWIND instances AS instance
78
79          // Build result of var in instance
80          CALL {
81              WITH vars, instance
82              UNWIND vars AS var
83
84              // Find read variables
85              CALL {
```

```
86              WITH var, instance
87              WITH var.var as var, instance
88              MATCH readVar=(var)<-[:CORRESPONDING_NAME_DE
                    FINITION]-(elt:AN_IDENTIFIER)
89                  WHERE NOT EXISTS((elt:AN_IDENTIFIER)-[:C
                        ORRESPONDING_ASSIGNATION]->(:AN_ASSI
                        GNMENT_STATEMENT))
90                  AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN]-
                        >(:A_PARAMETER_ASSOCIATION)-[:CORRES
                        PONDING_PARAMETER_SPECIFICATION]->(:
                        AN_OUT_MODE))
91                  AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]
                        ->(:AN_OBJECT_RENAMING_DECLARATION))
92                  AND EXISTS ((elt)-[:CORRESPONDING_INSTAN
                        CIATION]->(instance))
93              WITH collect(readVar) as readArray
94              RETURN size(readArray) > 0 AS isReadInst
95              }
96
97              // Find written variables
98              CALL {
99              WITH var, instance
100             WITH var.var as var, instance
101             MATCH writeVar=(var)<-[:CORRESPONDING_NAME_D
                    EFINITION]-(elt:AN_IDENTIFIER)
102                 WHERE (EXISTS((elt:AN_IDENTIFIER)-[:CORR
                        ESPONDING_ASSIGNATION]->(:AN_ASSIGNM
                        ENT_STATEMENT))
103                 OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_
                        PARAMETER_ASSOCIATION)-[:CORRESPONDI
                        NG_PARAMETER_SPECIFICATION]->(:AN_OU
                        T_MODE))
104                 OR EXISTS ((elt)-[:IS_ENCLOSED_IN]->(:A_
                        PARAMETER_ASSOCIATION)-[:CORRESPONDI
                        NG_PARAMETER_SPECIFICATION]->(:AN_IN
                        _OUT_MODE)))
105                 AND NOT EXISTS ((elt)-[:IS_ENCLOSED_IN*]
                        ->(:AN_OBJECT_RENAMING_DECLARATION))
106                 AND EXISTS ((elt)-[:CORRESPONDING_INSTAN
                        CIATION]->(instance))
107             WITH collect(writeVar) as writeArray
108             RETURN size(writeArray) > 0 AS isWriteInst
```

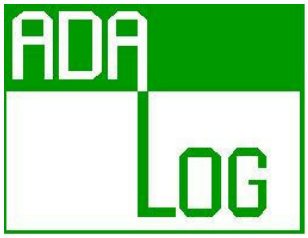
```
109             }
110
111             RETURN  var.var as Variable, isWriteInst,
                    isReadInst, var.isWriteGen AS
                    isWriteGen, var.isReadGen AS isReadGen
112         } // END: Build result of var in instance
113
114         return { filename: instance.filename, line:
                instance.line, column: instance.column } as
                Location, Variable, isWriteInst, isReadInst,
                isWriteGen, isReadGen
115     } // END: Find usage of every instantiations
116
117     RETURN cu as Compilation_Unit,
118         Location,
119         Variable,
120         isWriteInst OR isWriteGen AS isWrite,
121         isReadInst OR isReadGen AS isRead,
122         "instance" AS origin
123   } // END: Union Instance
124
125   RETURN Compilation_Unit,
126         Location,
127         Variable,
128         isWrite,
129         isRead,
130         origin
131 } // END: Aggregate all results for the final result
132
133 RETURN DISTINCT Compilation_Unit, Location, Variable,
      isWrite, isRead, origin
134   ORDER BY Location.filename, Location.line,
        Location.column;
```

Appendix C

Benchmark Results

C.1 Benchmark Report

This section includes the complete benchmark report generated from our measurement pipeline. It contains detailed per-project and aggregated results, tables, and plots complementing Chapter 3. We provide it in full for transparency and reproducibility.



Benchmark report

Benchmark Results

2025-12-14

Université de Caen Normandie, France
Adalog SAS, SIREN 527 695 704, France

Contents

1. Global Results

1.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	2 min 0 sec 600 ms	3 hr 0 min 19 sec 800 ms	1 min 27 sec 100 ms	1 min 28 sec 500 ms
overheadPopulating		46 min 3 sec 100 ms		
Relative Overhead (0 is better)	38.48%	15,490.42%	0.00%	1.57%
analysisTime	8 min 37 sec 500 ms	44 sec 90 ms	11 min 35 sec 800 ms	11 min 26 sec 1,000 ms
Analysis Relative Speed (0 is better)	1,073.80%	0.00%	1,478.05%	1,458.06%
executionTime	10 min 38 sec 200 ms	3 hr 47 min 6 sec 1,000 ms	13 min 2 sec 900 ms	12 min 55 sec 400 ms
Execution Relative Speed (0 is better)	0.00%	2,035.29%	22.68%	21.51%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	17,782	16,142	18,793	18,793

Table 1: Global: All Projects

Number of projects: 134

Total lines of code: 2,643,887

Feature	Description	Value
Attr Access All	Number of uses of attribute ‘Access on any object.	54
Attr Address All	Number of uses of attribute ‘Address on any object.	36
Attr Unchecked Access All	Number of uses of attribute ‘Unchecked_Access on any object.	21
Decls Operators Overloaded	Number of declarations of overloaded operators.	36

Feature	Description	Value
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	2
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	74
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	65
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	10
Exceptions Declared	Number of exception declarations.	53
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	4
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	51
Generics Inst Private	Number of private generic instantiations.	40
Generics Inst Public	Number of public generic instantiations.	47
Generics Units All	Number of generic units (generic packages and subprograms).	59
Handlers Others All	Number of exception handlers using the others choice.	44
Handlers Others Null	Number of others exception handlers whose body is null.	14
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	9
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).	0
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	4
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0

Feature	Description	Value
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	2
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	3
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0
Local Exception	Number of exceptions that are locally handled.	8
Metrics Functions Called	Number of function call sites.	115
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	128
Metrics Procedures Called	Number of procedure call sites.	103
Metrics Statements All	Total number of executable statements.	125
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	125
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	56
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	5
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	3
Pragmas All	Total number of pragmas.	116
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	96
Protected Objects Declared	Number of protected object declarations.	9
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	43
Statements Abort	Number of abort statements.	1
Statements Accept	Number of accept statements.	9
Statements Conditional Entry Call	Number of conditional entry call statements.	2
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	12
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	12
Statements Raise All	Number of raise statements (all exceptions).	77
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	48
Statements Requeue	Number of requeue statements.	1

Feature	Description	Value
Statements Selective Accept	Number of selective accept statements (select).	3
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	2
Statements Timed Entry Call	Number of timed entry call statements.	4
Tasks Declared	Number of task declarations.	3
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	8
Type Usage Pos On Enum	Number of uses of attribute 'Pos on enumeration types.	38
Types Abstract	Number of abstract type declarations.	25
Types Access Subprogram	Number of access-to-subprogram type declarations.	34
Types Controlled	Number of controlled type declarations.	29
Types Derived	Number of derived type declarations.	56
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	69
Types With Discriminants	Number of type declarations with discriminants.	55

Table 2: Global: All Projects - Language feature usage (projects count)

Feature	Description	Value
Attr Access All	Number of uses of attribute 'Access on any object.	8,302
Attr Address All	Number of uses of attribute 'Address on any object.	1,145
Attr Unchecked Access All	Number of uses of attribute 'Unchecked_Access on any object.	560
Decls Operators Overloaded	Number of declarations of overloaded operators.	473
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	1
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	7

Feature	Description	Value
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	3
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	4
Exceptions Declared	Number of exception declarations.	482
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	14
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	553
Generics Inst Private	Number of private generic instantiations.	256
Generics Inst Public	Number of public generic instantiations.	992
Generics Units All	Number of generic units (generic packages and subprograms).	640
Handlers Others All	Number of exception handlers using the others choice.	1,402
Handlers Others Null	Number of others exception handlers whose body is null.	34
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	869
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).	0
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	16
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	3

Feature	Description	Value
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	6
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0
Local Exception	Number of exceptions that are locally handled.	23
Metrics Functions Called	Number of function call sites.	14,845
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	78,296
Metrics Procedures Called	Number of procedure call sites.	13,672
Metrics Statements All	Total number of executable statements.	130,394
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	11,896
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	2,366
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	18
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	4
Pragmas All	Total number of pragmas.	18,818
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	4,337
Protected Objects Declared	Number of protected object declarations.	22
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	469
Statements Abort	Number of abort statements.	1
Statements Accept	Number of accept statements.	50
Statements Conditional Entry Call	Number of conditional entry call statements.	2
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	69
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	113
Statements Raise All	Number of raise statements (all exceptions).	2,889
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	921

Feature	Description	Value
Statements Requeue	Number of requeue statements.	23
Statements Selective Accept	Number of selective accept statements (select).	12
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	6
Statements Timed Entry Call	Number of timed entry call statements.	12
Tasks Declared	Number of task declarations.	6
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	24
Type Usage Pos On Enum	Number of uses of attribute 'Pos on enumeration types.	1,261
Types Abstract	Number of abstract type declarations.	166
Types Access Subprogram	Number of access-to-subprogram type declarations.	1,319
Types Controlled	Number of controlled type declarations.	275
Types Derived	Number of derived type declarations.	780
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	1,304
Types With Discriminants	Number of type declarations with discriminants.	397

Table 3: Global: All Projects - Language feature usage (sum of values)

1.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	1 min 1 sec 900 ms	23 min 31 sec 100 ms	57 sec 1,000 ms	59 sec 300 ms
overheadPopulating		14 min 16 sec 900 ms		
Relative Overhead (0 is better)	6.75%	3,811.95%	0.00%	2.24%
analysisTime	1 min 6 sec 100 ms	34 sec 900 ms	1 min 11 sec 200 ms	1 min 9 sec 300 ms
Analysis Relative Speed (0 is better)	89.48%	0.00%	103.94%	98.45%
executionTime	2 min 8 sec 10 ms	38 min 22 sec 900 ms	2 min 9 sec 100 ms	2 min 8 sec 500 ms

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Execution Relative Speed (0 is better)	0.00%	1,698.97%	0.89%	0.40%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	2,288	1,924	2,558	2,558

Table 4: Global: Small Projects (0-10k LoC)

Number of projects: 118

Total lines of code: 182,706

Feature	Description	Value
Attr Access All	Number of uses of attribute 'Access on any object.	40
Attr Address All	Number of uses of attribute 'Address on any object.	28
Attr Unchecked Access All	Number of uses of attribute 'Unchecked_Access on any object.	11
Decls Operators Overloaded	Number of declarations of overloaded operators.	24
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	2
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	61
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	51
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	6
Exceptions Declared	Number of exception declarations.	39
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	2
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	39

Feature	Description	Value
Generics Inst Private	Number of private generic instantiations.	28
Generics Inst Public	Number of public generic instantiations.	32
Generics Units All	Number of generic units (generic packages and subprograms).	46
Handlers Others All	Number of exception handlers using the others choice.	32
Handlers Others Null	Number of others exception handlers whose body is null.	6
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	4
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).	0
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	1
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	0
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	3
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0
Local Exception	Number of exceptions that are locally handled.	4
Metrics Functions Called	Number of function call sites.	100
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	113
Metrics Procedures Called	Number of procedure call sites.	88
Metrics Statements All	Total number of executable statements.	110
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	110
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	41
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	1

Feature	Description	Value
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	2
Pragmas All	Total number of pragmas.	101
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	82
Protected Objects Declared	Number of protected object declarations.	3
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	33
Statements Abort	Number of abort statements.	1
Statements Accept	Number of accept statements.	5
Statements Conditional Entry Call	Number of conditional entry call statements.	1
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	8
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	7
Statements Raise All	Number of raise statements (all exceptions).	62
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	34
Statements Requeue	Number of requeue statements.	0
Statements Selective Accept	Number of selective accept statements (select).	2
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	2
Statements Timed Entry Call	Number of timed entry call statements.	3
Tasks Declared	Number of task declarations.	2
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	5
Type Usage Pos On Enum	Number of uses of attribute ‘Pos on enumeration types.	24
Types Abstract	Number of abstract type declarations.	16
Types Access Subprogram	Number of access-to-subprogram type declarations.	24
Types Controlled	Number of controlled type declarations.	22
Types Derived	Number of derived type declarations.	43
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	57

Feature	Description	Value
Types With Discriminants	Number of type declarations with discriminants.	42

Table 5: Global: Small Projects (0-10k LoC) - Language feature usage (projects count)

Feature	Description	Value
Attr Access All	Number of uses of attribute ‘Access on any object.	492
Attr Address All	Number of uses of attribute ‘Address on any object.	334
Attr Unchecked Access All	Number of uses of attribute ‘Unchecked_Access on any object.	52
Decls Operators Overloaded	Number of declarations of overloaded operators.	119
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	1
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	7
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	2
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	4
Exceptions Declared	Number of exception declarations.	182
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	10
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	201
Generics Inst Private	Number of private generic instantiations.	112
Generics Inst Public	Number of public generic instantiations.	184
Generics Units All	Number of generic units (generic packages and subprograms).	226
Handlers Others All	Number of exception handlers using the others choice.	193
Handlers Others Null	Number of others exception handlers whose body is null.	9

Feature	Description	Value
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	4
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).	0
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	1
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	0
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	6
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0
Local Exception	Number of exceptions that are locally handled.	4
Metrics Functions Called	Number of function call sites.	3,969
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	15,982
Metrics Procedures Called	Number of procedure call sites.	3,074
Metrics Statements All	Total number of executable statements.	40,343
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	4,355
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	867
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	7
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	2
Pragmas All	Total number of pragmas.	3,762
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	882
Protected Objects Declared	Number of protected object declarations.	8

Feature	Description	Value
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	340
Statements Abort	Number of abort statements.	1
Statements Accept	Number of accept statements.	40
Statements Conditional Entry Call	Number of conditional entry call statements.	1
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	41
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	74
Statements Raise All	Number of raise statements (all exceptions).	963
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	248
Statements Requeue	Number of requeue statements.	0
Statements Selective Accept	Number of selective accept statements (select).	11
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	6
Statements Timed Entry Call	Number of timed entry call statements.	5
Tasks Declared	Number of task declarations.	5
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	21
Type Usage Pos On Enum	Number of uses of attribute ‘Pos on enumeration types.	301
Types Abstract	Number of abstract type declarations.	56
Types Access Subprogram	Number of access-to-subprogram type declarations.	126
Types Controlled	Number of controlled type declarations.	80
Types Derived	Number of derived type declarations.	217
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	447
Types With Discriminants	Number of type declarations with discriminants.	139

Table 6: Global: Small Projects (0-10k LoC) - Language feature usage (sum of values)

1.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	15 sec 90 ms	6 min 30 sec 300 ms	7 sec 1,000 ms	8 sec 20 ms
overheadPopulat-ing		3 min 8 sec 800 ms		
Relative Overhead (0 is better)	88.82%	7,144.95%	0.00%	0.33%
analysisTime	25 sec 900 ms	4 sec 100 ms	6 sec 600 ms	6 sec 500 ms
Analysis Relative Speed (0 is better)	524.46%	0.00%	59.07%	55.85%
executionTime	40 sec 1,000 ms	9 min 43 sec 300 ms	14 sec 600 ms	14 sec 500 ms
Execution Relative Speed (0 is better)	183.01%	3,927.11%	0.74%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	1,560	1,262	1,612	1,612

Table 7: Global: Medium Projects (10-30k LoC)

Number of projects: 9

Total lines of code: 122,288

Feature	Description	Value
Attr Access All	Number of uses of attribute ‘Access on any object.	8
Attr Address All	Number of uses of attribute ‘Address on any object.	3
Attr Unchecked Access All	Number of uses of attribute ‘Unchecked_Access on any object.	5
Decls Operators Overloaded	Number of declarations of overloaded opera-tors.	7
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	0
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	8

Feature	Description	Value
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	8
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	2
Exceptions Declared	Number of exception declarations.	9
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	2
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	7
Generics Inst Private	Number of private generic instantiations.	7
Generics Inst Public	Number of public generic instantiations.	9
Generics Units All	Number of generic units (generic packages and subprograms).	8
Handlers Others All	Number of exception handlers using the others choice.	7
Handlers Others Null	Number of others exception handlers whose body is null.	3
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	1
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).	0
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	0
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	2
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	0
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0

Feature	Description	Value
Local Exception	Number of exceptions that are locally handled.	3
Metrics Functions Called	Number of function call sites.	9
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	9
Metrics Procedures Called	Number of procedure call sites.	9
Metrics Statements All	Total number of executable statements.	9
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	9
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	9
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	1
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	1
Pragmas All	Total number of pragmas.	9
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	8
Protected Objects Declared	Number of protected object declarations.	2
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	5
Statements Abort	Number of abort statements.	0
Statements Accept	Number of accept statements.	2
Statements Conditional Entry Call	Number of conditional entry call statements.	1
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	3
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	3
Statements Raise All	Number of raise statements (all exceptions).	9
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	9
Statements Requeue	Number of requeue statements.	0
Statements Selective Accept	Number of selective accept statements (select).	0
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	0
Statements Timed Entry Call	Number of timed entry call statements.	0

Feature	Description	Value
Tasks Declared	Number of task declarations.	0
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	1
Type Usage Pos On Enum	Number of uses of attribute ‘Pos on enumeration types.	8
Types Abstract	Number of abstract type declarations.	4
Types Access Subprogram	Number of access-to-subprogram type declarations.	4
Types Controlled	Number of controlled type declarations.	2
Types Derived	Number of derived type declarations.	8
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	7
Types With Discriminants	Number of type declarations with discriminants.	8

Table 8: Global: Medium Projects (10-30k LoC) - Language feature usage (projects count)

Feature	Description	Value
Attr Access All	Number of uses of attribute ‘Access on any object.	5,683
Attr Address All	Number of uses of attribute ‘Address on any object.	60
Attr Unchecked Access All	Number of uses of attribute ‘Unchecked_Access on any object.	347
Decls Operators Overloaded	Number of declarations of overloaded operators.	156
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	0
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	3
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	2
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	4
Exceptions Declared	Number of exception declarations.	254

Feature	Description	Value
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	4
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	134
Generics Inst Private	Number of private generic instantiations.	38
Generics Inst Public	Number of public generic instantiations.	84
Generics Units All	Number of generic units (generic packages and subprograms).	104
Handlers Others All	Number of exception handlers using the others choice.	84
Handlers Others Null	Number of others exception handlers whose body is null.	6
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	1
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).	0
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	0
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	3
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	0
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0
Local Exception	Number of exceptions that are locally handled.	18
Metrics Functions Called	Number of function call sites.	1,649
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	10,823
Metrics Procedures Called	Number of procedure call sites.	2,774
Metrics Statements All	Total number of executable statements.	26,965

Feature	Description	Value
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	2,120
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	618
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	4
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	2
Pragmas All	Total number of pragmas.	1,502
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	213
Protected Objects Declared	Number of protected object declarations.	3
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	46
Statements Abort	Number of abort statements.	0
Statements Accept	Number of accept statements.	7
Statements Conditional Entry Call	Number of conditional entry call statements.	1
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	18
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	8
Statements Raise All	Number of raise statements (all exceptions).	820
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	87
Statements Requeue	Number of requeue statements.	0
Statements Selective Accept	Number of selective accept statements (select).	0
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	0
Statements Timed Entry Call	Number of timed entry call statements.	0
Tasks Declared	Number of task declarations.	0
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	1
Type Usage Pos On Enum	Number of uses of attribute ‘Pos on enumeration types.	181
Types Abstract	Number of abstract type declarations.	36

Feature	Description	Value
Types Access Subprogram	Number of access-to-subprogram type declarations.	124
Types Controlled	Number of controlled type declarations.	2
Types Derived	Number of derived type declarations.	90
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	332
Types With Discriminants	Number of type declarations with discriminants.	82

Table 9: Global: Medium Projects (10-30k LoC) - Language feature usage (sum of values)

1.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	43 sec 700 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)	106.43%	50,652.10%	0.00%	0.19%
analysisTime	7 min 5 sec 500 ms	5 sec 50 ms	10 min 18 sec 0 ms	10 min 11 sec 200 ms
Analysis Relative Speed (0 is better)	8,332.93%	0.00%	12,147.78%	12,013.74%
executionTime	7 min 49 sec 200 ms	2 hr 59 min 0 sec 800 ms	10 min 39 sec 200 ms	10 min 32 sec 400 ms
Execution Relative Speed (0 is better)	0.00%	2,189.29%	36.23%	34.80%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	13,934	12,956	14,623	14,623

Table 10: Global: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

Feature	Description	Value
Attr Access All	Number of uses of attribute ‘Access on any object.	6
Attr Address All	Number of uses of attribute ‘Address on any object.	5
Attr Unchecked Access All	Number of uses of attribute ‘Unchecked_Access on any object.	5
Decls Operators Overloaded	Number of declarations of overloaded operators.	5
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	0
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	5
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	6
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	2
Exceptions Declared	Number of exception declarations.	5
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	0
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	5
Generics Inst Private	Number of private generic instantiations.	5
Generics Inst Public	Number of public generic instantiations.	6
Generics Units All	Number of generic units (generic packages and subprograms).	5
Handlers Others All	Number of exception handlers using the others choice.	5
Handlers Others Null	Number of others exception handlers whose body is null.	5
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	4
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion con-	0

Feature	Description	Value
	verting System.Address to an access type (short form).	
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	3
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	0
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	0
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0
Local Exception	Number of exceptions that are locally handled.	1
Metrics Functions Called	Number of function call sites.	6
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	6
Metrics Procedures Called	Number of procedure call sites.	6
Metrics Statements All	Total number of executable statements.	6
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	6
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	6
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	3
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	0
Pragmas All	Total number of pragmas.	6
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	6
Protected Objects Declared	Number of protected object declarations.	4
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	5
Statements Abort	Number of abort statements.	0
Statements Accept	Number of accept statements.	2
Statements Conditional Entry Call	Number of conditional entry call statements.	0
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	1

Feature	Description	Value
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	2
Statements Raise All	Number of raise statements (all exceptions).	6
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	5
Statements Requeue	Number of requeue statements.	1
Statements Selective Accept	Number of selective accept statements (select).	1
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	0
Statements Timed Entry Call	Number of timed entry call statements.	1
Tasks Declared	Number of task declarations.	1
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	2
Type Usage Pos On Enum	Number of uses of attribute ‘Pos on enumeration types.	6
Types Abstract	Number of abstract type declarations.	5
Types Access Subprogram	Number of access-to-subprogram type declarations.	6
Types Controlled	Number of controlled type declarations.	5
Types Derived	Number of derived type declarations.	5
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	5
Types With Discriminants	Number of type declarations with discriminants.	5

Table 11: Global: Large Projects (30k+ LoC) - Language feature usage (projects count)

Feature	Description	Value
Attr Access All	Number of uses of attribute ‘Access on any object.	2,127
Attr Address All	Number of uses of attribute ‘Address on any object.	751
Attr Unchecked Access All	Number of uses of attribute ‘Unchecked_Access on any object.	161
Decls Operators Overloaded	Number of declarations of overloaded operators.	198

Feature	Description	Value
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).	0
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).	7
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).	0
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).	3
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.	4
Exceptions Declared	Number of exception declarations.	46
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).	0
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).	218
Generics Inst Private	Number of private generic instantiations.	106
Generics Inst Public	Number of public generic instantiations.	724
Generics Units All	Number of generic units (generic packages and subprograms).	310
Handlers Others All	Number of exception handlers using the others choice.	1,125
Handlers Others Null	Number of others exception handlers whose body is null.	19
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).	864
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).	0
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).	15
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).	0

Feature	Description	Value
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).	0
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.	0
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.	0
Local Exception	Number of exceptions that are locally handled.	1
Metrics Functions Called	Number of function call sites.	9,227
Metrics Objects All	Number of object declarations or usages (rough measure of data size).	51,491
Metrics Procedures Called	Number of procedure call sites.	7,824
Metrics Statements All	Total number of executable statements.	63,086
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).	5,421
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).	881
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).	7
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).	0
Pragmas All	Total number of pragmas.	13,554
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.	3,242
Protected Objects Declared	Number of protected object declarations.	11
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).	83
Statements Abort	Number of abort statements.	0
Statements Accept	Number of accept statements.	3
Statements Conditional Entry Call	Number of conditional entry call statements.	0
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).	10
Statements Delay Until	Number of delay until statements (delay until a specific time).	0
Statements Entry Call	Number of simple entry call statements.	31
Statements Raise All	Number of raise statements (all exceptions).	1,106
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.	586

Feature	Description	Value
Statements Requeue	Number of requeue statements.	23
Statements Selective Accept	Number of selective accept statements (select).	1
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.	0
Statements Timed Entry Call	Number of timed entry call statements.	7
Tasks Declared	Number of task declarations.	1
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).	2
Type Usage Pos On Enum	Number of uses of attribute 'Pos on enumeration types.	779
Types Abstract	Number of abstract type declarations.	74
Types Access Subprogram	Number of access-to-subprogram type declarations.	1,069
Types Controlled	Number of controlled type declarations.	193
Types Derived	Number of derived type declarations.	473
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.	525
Types With Discriminants	Number of type declarations with discriminants.	176

Table 12: Global: Large Projects (30k+ LoC) - Language feature usage (sum of values)

2. Results by Rules

2.1. Summary

2.1.1. Analysis Time

2.1.1.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	10 min 0 sec 900 ms	3 sec 400 ms	51 sec 900 ms	52 sec 300 ms	2,676,888	147
Abstract Type Declarations	9 min 29 sec 700 ms	4 sec 80 ms	51 sec 900 ms	52 sec 1,000 ms	2,676,888	147

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Blocks	9 min 46 sec 900 ms	2 sec 100 ms	52 sec 200 ms	52 sec 200 ms	2,676,888	147
Constructors	9 min 13 sec 10 ms	7 sec 700 ms	1 min 7 sec 300 ms	1 min 7 sec 200 ms	2,664,679	145
Enumeration Representation Clauses	9 min 36 sec 1,000 ms	1 sec 90 ms	41 sec 900 ms	42 sec 500 ms	2,655,240	135
Renamings	9 min 53 sec 500 ms	1 sec 1,000 ms	52 sec 90 ms	52 sec 300 ms	2,676,888	147
Slices	9 min 56 sec 400 ms	1 sec 700 ms	16 min 29 sec 200 ms	16 min 21 sec 800 ms	2,664,679	145
Too Many Parents	9 min 56 sec 400 ms	2 sec 300 ms	58 sec 800 ms	59 sec 700 ms	2,676,888	147
Variable Usage	3 min 48 sec 500 ms	1 min 5 sec 40 ms			655,191	140

Table 13: Analysis Time: All Projects

2.1.1.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	1 min 58 sec 10 ms	2 sec 600 ms	38 sec 500 ms	39 sec 70 ms	204,354	130
Abstract Type Declarations	1 min 55 sec 100 ms	2 sec 1,000 ms	38 sec 500 ms	39 sec 600 ms	204,354	130
Blocks	1 min 56 sec 300 ms	1 sec 400 ms	38 sec 700 ms	39 sec 3 ms	204,354	130
Constructors	1 min 56 sec 20 ms	6 sec 200 ms	43 sec 100 ms	43 sec 500 ms	203,498	129
Enumeration Representation Clauses	1 min 44 sec 100 ms	900 ms	28 sec 500 ms	29 sec 200 ms	182,706	118
Renamings	1 min 55 sec 700 ms	1 sec 300 ms	38 sec 700 ms	39 sec 40 ms	204,354	130

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Slices	1 min 58 sec 700 ms	1 sec 100 ms	48 sec 200 ms	48 sec 800 ms	203,498	129
Too Many Parents	1 min 59 sec 800 ms	1 sec 600 ms	41 sec 700 ms	42 sec 700 ms	204,354	130
Variable Usage	1 min 17 sec 90 ms	58 sec 500 ms			199,948	128

Table 14: Analysis Time: Small Projects (0-10k LoC)

2.1.1.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	25 sec 700 ms	300 ms	5 sec 500 ms	5 sec 500 ms	133,641	10
Abstract Type Declarations	18 sec 200 ms	500 ms	5 sec 600 ms	5 sec 600 ms	133,641	10
Blocks	22 sec 800 ms	300 ms	5 sec 600 ms	5 sec 500 ms	133,641	10
Constructors	14 sec 800 ms	800 ms	6 sec 600 ms	6 sec 500 ms	122,288	9
Enumeration Representation Clauses	25 sec 300 ms	80 ms	5 sec 500 ms	5 sec 500 ms	133,641	10
Renamings	25 sec 400 ms	300 ms	5 sec 700 ms	5 sec 600 ms	133,641	10
Slices	31 sec 600 ms	200 ms	8 sec 1,000 ms	8 sec 900 ms	122,288	9
Too Many Parents	25 sec 400 ms	300 ms	6 sec 300 ms	6 sec 300 ms	133,641	10
Variable Usage	1 min 29 sec 100 ms	4 sec 200 ms			106,623	8

Table 15: Analysis Time: Medium Projects (10-30k LoC)

2.1.1.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	7 min 37 sec 200 ms	400 ms	7 sec 900 ms	7 sec 800 ms	2,338,893	7
Abstract Type Declarations	7 min 16 sec 400 ms	600 ms	7 sec 900 ms	7 sec 800 ms	2,338,893	7
Blocks	7 min 27 sec 800 ms	500 ms	7 sec 900 ms	7 sec 600 ms	2,338,893	7
Constructors	7 min 2 sec 200 ms	700 ms	17 sec 600 ms	17 sec 200 ms	2,338,893	7
Enumeration Representation Clauses	7 min 27 sec 500 ms	80 ms	7 sec 900 ms	7 sec 800 ms	2,338,893	7
Renamings	7 min 32 sec 400 ms	300 ms	7 sec 800 ms	7 sec 600 ms	2,338,893	7
Slices	7 min 26 sec 100 ms	400 ms	15 min 31 sec 1,000 ms	15 min 24 sec 90 ms	2,338,893	7
Too Many Parents	7 min 31 sec 300 ms	300 ms	10 sec 800 ms	10 sec 700 ms	2,338,893	7
Variable Usage	1 min 2 sec 200 ms	2 sec 300 ms			348,620	4

Table 16: Analysis Time: Large Projects (30k+ LoC)

2.1.2. Parsing Overhead

2.1.2.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	5 sec 700 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms	2,676,888	147
Abstract Type Declarations	37 sec 1,000 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms	2,676,888	147

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Blocks	15 sec 700 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms	2,676,888	147
Construc-tors	44 sec 50 ms	3 hr 2 min 39 sec 500 ms	2 min 25 sec 500 ms	2 min 25 sec 50 ms	2,664,679	145
Enumera-tion Repre-sentation Clauses	4 sec 500 ms	3 hr 0 min 47 sec 900 ms	2 min 12 sec 300 ms	2 min 11 sec 800 ms	2,655,240	135
Renamings	7 sec 700 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms	2,676,888	147
Slices	34 sec 600 ms	3 hr 2 min 39 sec 500 ms	2 min 25 sec 500 ms	2 min 25 sec 50 ms	2,664,679	145
Too Many Parents	3 sec 100 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms	2,676,888	147
Variable Us-age	3 min 57 sec 300 ms	1 hr 8 min 6 sec 200 ms			655,191	140

Table 17: Parsing Overhead: All Projects

2.1.2.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort State-ments	5 sec 700 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms	204,354	130
Abstract Type Decla-rations	8 sec 600 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms	204,354	130
Blocks	7 sec 500 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms	204,354	130
Construc-tors	7 sec 70 ms	25 min 50 sec 900 ms	1 min 56 sec 400 ms	1 min 55 sec 800 ms	203,498	129
Enumera-tion Repre-sentation Clauses	4 sec 500 ms	23 min 31 sec 100 ms	1 min 42 sec 200 ms	1 min 41 sec 600 ms	182,706	118

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Renamings	7 sec 700 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms	204,354	130
Slices	15 sec 1,000 ms	25 min 50 sec 900 ms	1 min 56 sec 400 ms	1 min 55 sec 800 ms	203,498	129
Too Many Parents	3 sec 100 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms	204,354	130
Variable Usage	1 min 59 sec 500 ms	25 min 36 sec 200 ms			199,948	128

Table 18: Parsing Overhead: Small Projects (0-10k LoC)

2.1.2.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms	133,641	10
Abstract Type Declarations	8 sec 200 ms	6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms	133,641	10
Blocks	3 sec 100 ms	6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms	133,641	10
Constructors	9 sec 500 ms	6 min 30 sec 300 ms	7 sec 1,000 ms	8 sec 20 ms	122,288	9
Enumeration Representation Clauses		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms	133,641	10
Renamings		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms	133,641	10
Slices	7 sec 500 ms	6 min 30 sec 300 ms	7 sec 1,000 ms	8 sec 20 ms	122,288	9
Too Many Parents		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms	133,641	10
Variable Usage	19 sec 200 ms	6 min 15 sec 90 ms			106,623	8

Table 19: Parsing Overhead: Medium Projects (10-30k LoC)

2.1.2.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Abstract Type Declarations	21 sec 200 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Blocks	5 sec 60 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Constructors	27 sec 500 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Enumeration Representation Clauses		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Renamings		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Slices	11 sec 100 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Too Many Parents		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms	2,338,893	7
Variable Usage	1 min 38 sec 600 ms	36 min 14 sec 900 ms			348,620	4

Table 20: Parsing Overhead: Large Projects (30k+ LoC)

2.1.3. Issued Messages

2.1.3.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	2	1	2	2	2,676,888	147

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abstract Type Declarations	211	162	221	221	2,676,888	147
Blocks	8,322	7,896	8,529	8,529	2,676,888	147
Constructors	454	235	423	423	2,664,679	145
Enumeration Representation Clauses	75	71	78	78	2,655,240	135
Renamings	3,332	2,573	3,402	3,402	2,676,888	147
Slices	5,679	5,481	5,945	5,945	2,664,679	145
Too Many Parents	24	39	518	518	2,676,888	147
Variable Usage	22,134	11,892	0	0	655,191	140

Table 21: Issued Messages: All Projects

2.1.3.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	1	0	1	1	204,354	130
Abstract Type Declarations	71	59	71	71	204,354	130
Blocks	863	689	948	948	204,354	130
Constructors	129	118	134	134	203,498	129
Enumeration Representation Clauses	60	56	63	63	182,706	118
Renamings	567	472	599	599	204,354	130
Slices	848	776	920	920	203,498	129
Too Many Parents	4	8	84	84	204,354	130
Variable Usage	8,035	4,927	0	0	199,948	128

Table 22: Issued Messages: Small Projects (0-10k LoC)

2.1.3.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	0	0	0	0	133,641	10
Abstract Type Declarations	51	38	51	51	133,641	10
Blocks	462	338	476	476	133,641	10
Constructors	63	25	50	50	122,288	9
Enumeration Representation Clauses	2	2	2	2	133,641	10
Renamings	477	391	485	485	133,641	10
Slices	566	528	575	575	122,288	9
Too Many Parents	1	2	36	36	133,641	10
Variable Usage	5,849	4,227	0	0	106,623	8

Table 23: Issued Messages: Medium Projects (10-30k LoC)

2.1.3.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores	Number Of Line Of Codes	Number Of Projects
Abort Statements	1	1	1	1	2,338,893	7
Abstract Type Declarations	89	65	99	99	2,338,893	7
Blocks	6,997	6,869	7,105	7,105	2,338,893	7
Constructors	262	92	239	239	2,338,893	7
Enumeration Representation Clauses	13	13	13	13	2,338,893	7
Renamings	2,288	1,710	2,318	2,318	2,338,893	7
Slices	4,265	4,177	4,450	4,450	2,338,893	7
Too Many Parents	19	29	398	398	2,338,893	7
Variable Usage	8,250	2,738	0	0	348,620	4

Table 24: Issued Messages: Large Projects (30k+ LoC)

2.2. Abort Statements

2.2.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	5 sec 700 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms
overheadPopulating		47 min 57 sec 10 ms		
Relative Overhead (0 is better)	0.00%	243,292.95%	2,486.95%	2,478.29%
analysisTime	10 min 0 sec 900 ms	3 sec 400 ms	51 sec 900 ms	52 sec 300 ms
Analysis Relative Speed (0 is better)	17,680.18%	0.00%	1,436.29%	1,448.52%
executionTime	10 min 6 sec 600 ms	7 hr 42 min 13 sec 900 ms	3 min 19 sec 300 ms	3 min 19 sec 200 ms
Execution Relative Speed (0 is better)	204.51%	13,821.73%	0.04%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	2	1	2	2

Table 25: Abort Statements: All Projects

Number of projects: 147

Total lines of code: 2,676,888

2.2.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	5 sec 700 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms
overheadPopulating		15 min 54 sec 600 ms		
Relative Overhead (0 is better)	0.00%	43,892.31%	1,958.75%	1,948.92%
analysisTime	1 min 58 sec 10 ms	2 sec 600 ms	38 sec 500 ms	39 sec 70 ms

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Analysis Relative Speed (0 is better)	4,353.40%	0.00%	1,353.36%	1,374.24%
executionTime	2 min 3 sec 700 ms	1 hr 23 min 34 sec 800 ms	2 min 35 sec 800 ms	2 min 35 sec 800 ms
Execution Relative Speed (0 is better)	0.00%	3,953.71%	25.93%	25.93%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	1	0	1	1

Table 26: Abort Statements: Small Projects (0-10k LoC)

Number of projects: 130

Total lines of code: 204,354

2.2.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms
overheadPopulating		3 min 25 sec 3 ms		
Relative Overhead (0 is better)		6,876.09%	0.00%	0.30%
analysisTime	25 sec 700 ms	300 ms	5 sec 500 ms	5 sec 500 ms
Analysis Relative Speed (0 is better)	7,680.73%	0.00%	1,572.78%	1,557.64%
executionTime	25 sec 700 ms	20 min 47 sec 200 ms	14 sec 500 ms	14 sec 400 ms
Execution Relative Speed (0 is better)	78.02%	8,537.05%	0.16%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	0	0	0	0

Table 27: Abort Statements: Medium Projects (10-30k LoC)

Number of projects: 10

Total lines of code: 133,641

2.2.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)		50,652.10%	0.00%	0.19%
analysisTime	7 min 37 sec 200 ms	400 ms	7 sec 900 ms	7 sec 800 ms
Analysis Relative Speed (0 is better)	114,361.72%	0.00%	1,873.56%	1,851.03%
executionTime	7 min 37 sec 200 ms	5 hr 57 min 51 sec 900 ms	29 sec 40 ms	28 sec 1,000 ms
Execution Relative Speed (0 is better)	1,477.32%	73,975.16%	0.17%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	1	1	1	1

Table 28: Abort Statements: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.3. Abstract Type Declarations

2.3.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	37 sec 1,000 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms
overheadPopulating		47 min 57 sec 10 ms		
Relative Overhead (0 is better)	0.00%	36,429.25%	288.26%	286.96%

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
analysisTime	9 min 29 sec 700 ms	4 sec 80 ms	51 sec 900 ms	52 sec 1,000 ms
Analysis Relative Speed (0 is better)	13,879.28%	0.00%	1,174.58%	1,200.10%
executionTime	10 min 7 sec 700 ms	7 hr 42 min 14 sec 600 ms	3 min 19 sec 300 ms	3 min 19 sec 900 ms
Execution Relative Speed (0 is better)	204.89%	13,814.87%	0.00%	0.27%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	211	162	221	221

Table 29: Abstract Type Declarations: All Projects

Number of projects: 147

Total lines of code: 2,676,888

2.3.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	8 sec 600 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms
overheadPopulating		15 min 54 sec 600 ms		
Relative Overhead (0 is better)	0.00%	28,905.73%	1,257.41%	1,250.93%
analysisTime	1 min 55 sec 100 ms	2 sec 1,000 ms	38 sec 500 ms	39 sec 600 ms
Analysis Relative Speed (0 is better)	3,759.72%	0.00%	1,189.29%	1,227.73%
executionTime	2 min 3 sec 800 ms	1 hr 23 min 35 sec 200 ms	2 min 35 sec 700 ms	2 min 36 sec 300 ms
Execution Relative Speed (0 is better)	0.00%	3,951.79%	25.82%	26.30%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Issued Messages	71	59	71	71

Table 30: Abstract Type Declarations: Small Projects (0-10k LoC)

Number of projects: 130

Total lines of code: 204,354

2.3.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	8 sec 200 ms	6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms
overheadPopulat-ing		3 min 25 sec 3 ms		
Relative Overhead (0 is better)	0.00%	7,549.45%	9.65%	9.98%
analysisTime	18 sec 200 ms	500 ms	5 sec 600 ms	5 sec 600 ms
Analysis Relative Speed (0 is better)	3,550.68%	0.00%	1,014.76%	1,013.43%
executionTime	26 sec 400 ms	20 min 47 sec 400 ms	14 sec 500 ms	14 sec 500 ms
Execution Relative Speed (0 is better)	81.89%	8,500.50%	0.00%	0.14%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	51	38	51	51

Table 31: Abstract Type Declarations: Medium Projects (10-30k LoC)

Number of projects: 10

Total lines of code: 133,641

2.3.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	21 sec 200 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)	0.06%	50,652.10%	0.00%	0.19%
analysisTime	7 min 16 sec 400 ms	600 ms	7 sec 900 ms	7 sec 800 ms
Analysis Relative Speed (0 is better)	73,462.93%	0.00%	1,235.15%	1,218.29%
executionTime	7 min 37 sec 500 ms	5 hr 57 min 52 sec 100 ms	29 sec 70 ms	29 sec 10 ms
Execution Relative Speed (0 is better)	1,476.99%	73,907.74%	0.21%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	89	65	99	99

Table 32: Abstract Type Declarations: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.4. Blocks

2.4.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	15 sec 700 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms
overheadPopulating		47 min 57 sec 10 ms		
Relative Overhead (0 is better)	0.00%	88,082.86%	837.27%	834.13%
analysisTime	9 min 46 sec 900 ms	2 sec 100 ms	52 sec 200 ms	52 sec 200 ms
Analysis Relative Speed (0 is better)	27,652.22%	0.00%	2,367.16%	2,366.21%

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
executionTime	10 min 2 sec 600 ms	7 hr 42 min 12 sec 700 ms	3 min 19 sec 500 ms	3 min 19 sec 30 ms
Execution Relative Speed (0 is better)	202.77%	13,833.92%	0.26%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	8,322	7,896	8,529	8,529

Table 33: Blocks: All Projects

Number of projects: 147

Total lines of code: 2,676,888

2.4.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	7 sec 500 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms
overheadPopulating		15 min 54 sec 600 ms		
Relative Overhead (0 is better)	0.00%	33,196.21%	1,458.19%	1,450.75%
analysisTime	1 min 56 sec 300 ms	1 sec 400 ms	38 sec 700 ms	39 sec 3 ms
Analysis Relative Speed (0 is better)	8,291.01%	0.00%	2,692.59%	2,714.48%
executionTime	2 min 3 sec 800 ms	1 hr 23 min 33 sec 600 ms	2 min 35 sec 1,000 ms	2 min 35 sec 700 ms
Execution Relative Speed (0 is better)	0.00%	3,949.41%	25.98%	25.78%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	863	689	948	948

Table 34: Blocks: Small Projects (0-10k LoC)

Number of projects: 130

Total lines of code: 204,354

2.4.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	3 sec 100 ms	6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms
overheadPopulating		3 min 25 sec 3 ms		
Relative Overhead (0 is better)	0.00%	19,754.46%	184.61%	185.46%
analysisTime	22 sec 800 ms	300 ms	5 sec 600 ms	5 sec 500 ms
Analysis Relative Speed (0 is better)	8,866.60%	0.00%	2,100.37%	2,063.67%
executionTime	25 sec 900 ms	20 min 47 sec 100 ms	14 sec 500 ms	14 sec 500 ms
Execution Relative Speed (0 is better)	79.35%	8,520.61%	0.46%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	462	338	476	476

Table 35: Blocks: Medium Projects (10-30k LoC)

Number of projects: 10

Total lines of code: 133,641

2.4.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	5 sec 60 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)	0.00%	212,209.03%	318.33%	319.12%
analysisTime	7 min 27 sec 800 ms	500 ms	7 sec 900 ms	7 sec 600 ms
Analysis Relative Speed (0 is better)	94,260.34%	0.00%	1,559.81%	1,511.34%

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
executionTime	7 min 32 sec 800 ms	5 hr 57 min 51 sec 1,000 ms	29 sec 30 ms	28 sec 800 ms
Execution Relative Speed (0 is better)	1,470.20%	74,352.13%	0.66%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	6,997	6,869	7,105	7,105

Table 36: Blocks: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.5. Constructors

2.5.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	44 sec 50 ms	3 hr 2 min 39 sec 500 ms	2 min 25 sec 500 ms	2 min 25 sec 50 ms
overheadPopulating		47 min 36 sec 700 ms		
Relative Overhead (0 is better)	0.00%	31,262.61%	230.34%	229.26%
analysisTime	9 min 13 sec 10 ms	7 sec 700 ms	1 min 7 sec 300 ms	1 min 7 sec 200 ms
Analysis Relative Speed (0 is better)	7,120.44%	0.00%	778.27%	777.40%
executionTime	9 min 57 sec 70 ms	7 hr 40 min 40 sec 200 ms	3 min 32 sec 800 ms	3 min 32 sec 300 ms
Execution Relative Speed (0 is better)	181.30%	12,922.48%	0.26%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	454	235	423	423

Table 37: Constructors: All Projects

Number of projects: 145

Total lines of code: 2,664,679

2.5.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	7 sec 70 ms	25 min 50 sec 900 ms	1 min 56 sec 400 ms	1 min 55 sec 800 ms
overheadPopulating		15 min 50 sec 500 ms		
Relative Overhead (0 is better)	0.00%	35,280.50%	1,546.11%	1,538.43%
analysisTime	1 min 56 sec 20 ms	6 sec 200 ms	43 sec 100 ms	43 sec 500 ms
Analysis Relative Speed (0 is better)	1,766.63%	0.00%	593.72%	599.35%
executionTime	2 min 3 sec 90 ms	1 hr 23 min 29 sec 20 ms	2 min 39 sec 500 ms	2 min 39 sec 300 ms
Execution Relative Speed (0 is better)	0.00%	3,969.51%	29.58%	29.42%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	129	118	134	134

Table 38: Constructors: Small Projects (0-10k LoC)

Number of projects: 129

Total lines of code: 203,498

2.5.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	9 sec 500 ms	6 min 30 sec 300 ms	7 sec 1,000 ms	8 sec 20 ms
overheadPopulating		3 min 8 sec 800 ms		
Relative Overhead (0 is better)	18.85%	7,144.95%	0.00%	0.33%

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
analysisTime	14 sec 800 ms	800 ms	6 sec 600 ms	6 sec 500 ms
Analysis Relative Speed (0 is better)	1,820.23%	0.00%	753.43%	745.20%
executionTime	24 sec 300 ms	19 min 18 sec 1,000 ms	14 sec 600 ms	14 sec 500 ms
Execution Relative Speed (0 is better)	67.12%	7,883.89%	0.25%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	63	25	50	50

Table 39: Constructors: Medium Projects (10-30k LoC)

Number of projects: 9

Total lines of code: 122,288

2.5.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	27 sec 500 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)	29.92%	50,652.10%	0.00%	0.19%
analysisTime	7 min 2 sec 200 ms	700 ms	17 sec 600 ms	17 sec 200 ms
Analysis Relative Speed (0 is better)	62,451.02%	0.00%	2,505.82%	2,453.48%
executionTime	7 min 29 sec 700 ms	5 hr 57 min 52 sec 200 ms	38 sec 700 ms	38 sec 400 ms
Execution Relative Speed (0 is better)	1,070.23%	55,773.52%	0.82%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	262	92	239	239

Table 40: Constructors: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.6. Enumeration Representation Clauses

2.6.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	4 sec 500 ms	3 hr 0 min 47 sec 900 ms	2 min 12 sec 300 ms	2 min 11 sec 800 ms
overheadPopulating		46 min 19 sec 300 ms		
Relative Overhead (0 is better)	0.00%	299,838.55%	2,811.45%	2,800.37%
analysisTime	9 min 36 sec 1,000 ms	1 sec 90 ms	41 sec 900 ms	42 sec 500 ms
Analysis Relative Speed (0 is better)	52,922.98%	0.00%	3,751.55%	3,805.78%
executionTime	9 min 41 sec 500 ms	7 hr 34 min 15 sec 500 ms	2 min 54 sec 200 ms	2 min 54 sec 300 ms
Execution Relative Speed (0 is better)	233.84%	15,547.30%	0.00%	0.05%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	75	71	78	78

Table 41: Enumeration Representation Clauses: All Projects

Number of projects: 135

Total lines of code: 2,655,240

2.6.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	4 sec 500 ms	23 min 31 sec 100 ms	1 min 42 sec 200 ms	1 min 41 sec 600 ms
overheadPopulating		14 min 16 sec 900 ms		

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Relative Overhead (0 is better)	0.00%	49,819.70%	2,149.16%	2,136.61%
analysisTime	1 min 44 sec 100 ms	900 ms	28 sec 500 ms	29 sec 200 ms
Analysis Relative Speed (0 is better)	11,196.89%	0.00%	2,997.08%	3,071.22%
executionTime	1 min 48 sec 700 ms	1 hr 15 min 36 sec 1,000 ms	2 min 10 sec 700 ms	2 min 10 sec 800 ms
Execution Relative Speed (0 is better)	0.00%	4,074.99%	20.30%	20.41%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	60	56	63	63

Table 42: Enumeration Representation Clauses: Small Projects (0-10k LoC)

Number of projects: 118

Total lines of code: 182,706

2.6.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms
overheadPopulating		3 min 25 sec 3 ms		
Relative Overhead (0 is better)		6,876.09%	0.00%	0.30%
analysisTime	25 sec 300 ms	80 ms	5 sec 500 ms	5 sec 500 ms
Analysis Relative Speed (0 is better)	30,111.45%	0.00%	6,441.77%	6,481.54%
executionTime	25 sec 300 ms	20 min 46 sec 900 ms	14 sec 400 ms	14 sec 500 ms
Execution Relative Speed (0 is better)	75.61%	8,547.32%	0.00%	0.42%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Issued Messages	2	2	2	2

Table 43: Enumeration Representation Clauses: Medium Projects (10-30k LoC)

Number of projects: 10

Total lines of code: 133,641

2.6.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)		50,652.10%	0.00%	0.19%
analysisTime	7 min 27 sec 500 ms	80 ms	7 sec 900 ms	7 sec 800 ms
Analysis Relative Speed (0 is better)	541,790.07%	0.00%	9,441.90%	9,288.52%
executionTime	7 min 27 sec 500 ms	5 hr 57 min 51 sec 600 ms	29 sec 30 ms	28 sec 900 ms
Execution Relative Speed (0 is better)	1,445.98%	74,076.43%	0.30%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	13	13	13	13

Table 44: Enumeration Representation Clauses: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.7. Renamings

2.7.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	7 sec 700 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms
overheadPopulating		47 min 57 sec 10 ms		
Relative Overhead (0 is better)	0.00%	179,735.08%	1,811.41%	1,805.02%
analysisTime	9 min 53 sec 500 ms	1 sec 1,000 ms	52 sec 90 ms	52 sec 300 ms
Analysis Relative Speed (0 is better)	30,289.37%	0.00%	2,567.13%	2,575.84%
executionTime	10 min 1 sec 200 ms	7 hr 42 min 12 sec 500 ms	3 min 19 sec 500 ms	3 min 19 sec 100 ms
Execution Relative Speed (0 is better)	201.90%	13,826.61%	0.16%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	3,332	2,573	3,402	3,402

Table 45: Renamings: All Projects

Number of projects: 147

Total lines of code: 2,676,888

2.7.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	7 sec 700 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms
overheadPopulating		15 min 54 sec 600 ms		
Relative Overhead (0 is better)	0.00%	32,404.47%	1,421.14%	1,413.88%
analysisTime	1 min 55 sec 700 ms	1 sec 300 ms	38 sec 700 ms	39 sec 40 ms

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Analysis Relative Speed (0 is better)	8,672.50%	0.00%	2,830.99%	2,860.82%
executionTime	2 min 3 sec 400 ms	1 hr 23 min 33 sec 500 ms	2 min 35 sec 900 ms	2 min 35 sec 800 ms
Execution Relative Speed (0 is better)	0.00%	3,963.14%	26.37%	26.24%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	567	472	599	599

Table 46: Renamings: Small Projects (0-10k LoC)

Number of projects: 130

Total lines of code: 204,354

2.7.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms
overheadPopulating		3 min 25 sec 3 ms		
Relative Overhead (0 is better)		6,876.09%	0.00%	0.30%
analysisTime	25 sec 400 ms	300 ms	5 sec 700 ms	5 sec 600 ms
Analysis Relative Speed (0 is better)	8,612.95%	0.00%	1,838.88%	1,812.57%
executionTime	25 sec 400 ms	20 min 47 sec 200 ms	14 sec 600 ms	14 sec 500 ms
Execution Relative Speed (0 is better)	74.66%	8,479.35%	0.34%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	477	391	485	485

Table 47: Renamings: Medium Projects (10-30k LoC)

Number of projects: 10

Total lines of code: 133,641

2.7.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)		50,652.10%	0.00%	0.19%
analysisTime	7 min 32 sec 400 ms	300 ms	7 sec 800 ms	7 sec 600 ms
Analysis Relative Speed (0 is better)	131,859.52%	0.00%	2,171.24%	2,128.46%
executionTime	7 min 32 sec 400 ms	5 hr 57 min 51 sec 900 ms	28 sec 900 ms	28 sec 800 ms
Execution Relative Speed (0 is better)	1,469.04%	74,368.89%	0.37%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	2,288	1,710	2,318	2,318

Table 48: Renamings: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.8. Slices

2.8.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	34 sec 600 ms	3 hr 2 min 39 sec 500 ms	2 min 25 sec 500 ms	2 min 25 sec 50 ms
overheadPopulating		47 min 36 sec 700 ms		
Relative Overhead (0 is better)	0.00%	39,812.21%	320.39%	319.02%

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
analysisTime	9 min 56 sec 400 ms	1 sec 700 ms	16 min 29 sec 200 ms	16 min 21 sec 800 ms
Analysis Relative Speed (0 is better)	34,721.36%	0.00%	57,653.67%	57,220.47%
executionTime	10 min 31 sec 40 ms	7 hr 40 min 34 sec 300 ms	18 min 54 sec 700 ms	18 min 46 sec 800 ms
Execution Relative Speed (0 is better)	0.00%	4,279.16%	79.82%	78.57%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	5,679	5,481	5,945	5,945

Table 49: Slices: All Projects

Number of projects: 145

Total lines of code: 2,664,679

2.8.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	15 sec 1,000 ms	25 min 50 sec 900 ms	1 min 56 sec 400 ms	1 min 55 sec 800 ms
overheadPopulating		15 min 50 sec 500 ms		
Relative Overhead (0 is better)	0.00%	15,540.28%	627.68%	624.28%
analysisTime	1 min 58 sec 700 ms	1 sec 100 ms	48 sec 200 ms	48 sec 800 ms
Analysis Relative Speed (0 is better)	10,445.89%	0.00%	4,184.44%	4,232.40%
executionTime	2 min 14 sec 700 ms	1 hr 23 min 23 sec 900 ms	2 min 44 sec 600 ms	2 min 44 sec 600 ms
Execution Relative Speed (0 is better)	0.00%	3,613.95%	22.18%	22.18%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Issued Messages	848	776	920	920

Table 50: Slices: Small Projects (0-10k LoC)

Number of projects: 129

Total lines of code: 203,498

2.8.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	7 sec 500 ms	6 min 30 sec 300 ms	7 sec 1,000 ms	8 sec 20 ms
overheadPopulat-ing		3 min 8 sec 800 ms		
Relative Overhead (0 is better)	0.00%	7,624.94%	6.63%	6.98%
analysisTime	31 sec 600 ms	200 ms	8 sec 1,000 ms	8 sec 900 ms
Analysis Relative Speed (0 is better)	13,805.31%	0.00%	3,853.02%	3,828.07%
executionTime	39 sec 70 ms	19 min 18 sec 500 ms	16 sec 1,000 ms	16 sec 900 ms
Execution Relative Speed (0 is better)	130.66%	6,738.57%	0.18%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	566	528	575	575

Table 51: Slices: Medium Projects (10-30k LoC)

Number of projects: 9

Total lines of code: 122,288

2.8.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	11 sec 100 ms	2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)	0.00%	96,386.76%	90.11%	90.47%
analysisTime	7 min 26 sec 100 ms	400 ms	15 min 31 sec 1,000 ms	15 min 24 sec 90 ms
Analysis Relative Speed (0 is better)	123,891.02%	0.00%	258,938.51%	256,741.85%
executionTime	7 min 37 sec 200 ms	5 hr 57 min 51 sec 900 ms	15 min 53 sec 100 ms	15 min 45 sec 300 ms
Execution Relative Speed (0 is better)	0.00%	4,596.04%	108.46%	106.74%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	4,265	4,177	4,450	4,450

Table 52: Slices: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.9. Too Many Parents

2.9.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	3 sec 100 ms	3 hr 3 min 8 sec 300 ms	2 min 27 sec 400 ms	2 min 26 sec 900 ms
overheadPopulating		47 min 57 sec 10 ms		
Relative Overhead (0 is better)	0.00%	445,729.10%	4,638.59%	4,622.72%
analysisTime	9 min 56 sec 400 ms	2 sec 300 ms	58 sec 800 ms	59 sec 700 ms
Analysis Relative Speed (0 is better)	26,081.25%	0.00%	2,480.92%	2,520.71%

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
executionTime	9 min 59 sec 600 ms	7 hr 42 min 12 sec 800 ms	3 min 26 sec 200 ms	3 min 26 sec 600 ms
Execution Relative Speed (0 is better)	190.81%	13,351.66%	0.00%	0.20%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	24	39	518	518

Table 53: Too Many Parents: All Projects

Number of projects: 147

Total lines of code: 2,676,888

2.9.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	3 sec 100 ms	25 min 51 sec 500 ms	1 min 57 sec 300 ms	1 min 56 sec 700 ms
overheadPopulating		15 min 54 sec 600 ms		
Relative Overhead (0 is better)	0.00%	80,481.83%	3,671.06%	3,653.05%
analysisTime	1 min 59 sec 800 ms	1 sec 600 ms	41 sec 700 ms	42 sec 700 ms
Analysis Relative Speed (0 is better)	7,264.41%	0.00%	2,466.77%	2,524.16%
executionTime	2 min 2 sec 900 ms	1 hr 23 min 33 sec 800 ms	2 min 39 sec 30 ms	2 min 39 sec 400 ms
Execution Relative Speed (0 is better)	0.00%	3,980.03%	29.41%	29.71%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	4	8	84	84

Table 54: Too Many Parents: Small Projects (0-10k LoC)

Number of projects: 130

Total lines of code: 204,354

2.9.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		6 min 58 sec 400 ms	8 sec 900 ms	8 sec 1,000 ms
overheadPopulating		3 min 25 sec 3 ms		
Relative Overhead (0 is better)		6,876.09%	0.00%	0.30%
analysisTime	25 sec 400 ms	300 ms	6 sec 300 ms	6 sec 300 ms
Analysis Relative Speed (0 is better)	8,078.15%	0.00%	1,919.27%	1,934.32%
executionTime	25 sec 400 ms	20 min 47 sec 200 ms	15 sec 200 ms	15 sec 300 ms
Execution Relative Speed (0 is better)	66.83%	8,106.87%	0.00%	0.48%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	1	2	36	36

Table 55: Too Many Parents: Medium Projects (10-30k LoC)

Number of projects: 10

Total lines of code: 133,641

2.9.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing		2 hr 30 min 18 sec 400 ms	21 sec 200 ms	21 sec 200 ms
overheadPopulating		28 min 37 sec 400 ms		
Relative Overhead (0 is better)		50,652.10%	0.00%	0.19%
analysisTime	7 min 31 sec 300 ms	300 ms	10 sec 800 ms	10 sec 700 ms
Analysis Relative Speed (0 is better)	131,981.24%	0.00%	3,057.80%	3,036.34%

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
executionTime	7 min 31 sec 300 ms	5 hr 57 min 51 sec 900 ms	31 sec 900 ms	31 sec 900 ms
Execution Relative Speed (0 is better)	1,314.33%	67,188.82%	0.10%	0.00%
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	19	29	398	398

Table 56: Too Many Parents: Large Projects (30k+ LoC)

Number of projects: 7

Total lines of code: 2,338,893

2.10. Variable Usage

2.10.1. All Projects

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	3 min 57 sec 300 ms	1 hr 8 min 6 sec 200 ms		
overheadPopulating		21 min 56 sec 600 ms		
Relative Overhead (0 is better)	0.00%	2,177.05%		
analysisTime	3 min 48 sec 500 ms	1 min 5 sec 40 ms		
Analysis Relative Speed (0 is better)	251.24%	0.00%		
executionTime	7 min 45 sec 700 ms	3 hr 1 min 10 sec 700 ms		
Execution Relative Speed (0 is better)	0.00%	2,234.16%		
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	22,134	11,892	0	0

Table 57: Variable Usage: All Projects

Number of projects: 140

Total lines of code: 655,191

2.10.2. Small Projects (0-10k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	1 min 59 sec 500 ms	25 min 36 sec 200 ms		
overheadPopulating		15 min 34 sec 800 ms		
Relative Overhead (0 is better)	0.00%	1,967.02%		
analysisTime	1 min 17 sec 90 ms	58 sec 500 ms		
Analysis Relative Speed (0 is better)	31.74%	0.00%		
executionTime	3 min 16 sec 600 ms	1 hr 23 min 20 sec 500 ms		
Execution Relative Speed (0 is better)	0.00%	2,443.01%		
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	8,035	4,927	0	0

Table 58: Variable Usage: Small Projects (0-10k LoC)

Number of projects: 128

Total lines of code: 199,948

2.10.3. Medium Projects (10-30k LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	19 sec 200 ms	6 min 15 sec 90 ms		
overheadPopulating		2 min 53 sec 500 ms		
Relative Overhead (0 is better)	0.00%	2,761.57%		

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
analysisTime	1 min 29 sec 100 ms	4 sec 200 ms		
Analysis Relative Speed (0 is better)	2,021.39%	0.00%		
executionTime	1 min 48 sec 300 ms	18 min 21 sec 300 ms		
Execution Relative Speed (0 is better)	0.00%	916.92%		
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0
Issued Messages	5,849	4,227	0	0

Table 59: Variable Usage: Medium Projects (10-30k LoC)

Number of projects: 8

Total lines of code: 106,623

2.10.4. Large Projects (30k+ LoC)

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
overheadParsing	1 min 38 sec 600 ms	36 min 14 sec 900 ms		
overheadPopulating		3 min 28 sec 300 ms		
Relative Overhead (0 is better)	0.00%	2,318.11%		
analysisTime	1 min 2 sec 200 ms	2 sec 300 ms		
Analysis Relative Speed (0 is better)	2,581.59%	0.00%		
executionTime	2 min 40 sec 800 ms	1 hr 19 min 28 sec 900 ms		
Execution Relative Speed (0 is better)	0.00%	2,865.98%		
Nb run fails	0	0	0	0
Nb project fails	0	0	0	0

Metric	Adactl	Cogralys	Gnatcheck 1cores	Gnatcheck 32cores
Issued Messages	8,250	2,738	0	0

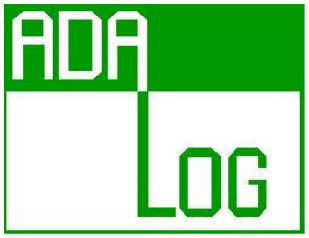
Table 60: Variable Usage: Large Projects (30k+ LoC)

Number of projects: 4

Total lines of code: 348,620

C.2 Import analysis

This section provides the import analysis used to interpret small-project behavior (clusters C1/C2) and library effects discussed in Section 3.2.2. The PDF presents the supporting breakdowns and cross-tool comparisons.



Benchmark report

Analysis of imports for project size < 10,000 LoC

2025-12-14

Université de Caen Normandie, France
Adalog SAS, SIREN 527 695 704, France

Contents

1. Mathematical Background

In this report, we use various statistical measures to analyze the data. Below are the key mathematical formulae used:

1.1. Pearson Correlation Coefficient

The correlation between analysis time and various metrics is calculated using the Pearson correlation coefficient formula:

$$r = \frac{n\sum xy - \sum x \sum y}{\sqrt{\left[n\sum x^2 - \left(\sum x\right)^2\right]\left[n\sum y^2 - \left(\sum y\right)^2\right]}}$$

Where:

- n is the number of data points (projects)
- $\sum xy$ is the sum of the products of paired data values
- $\sum x$ is the sum of the x values (analysis times)
- $\sum y$ is the sum of the y values (metric values)
- $\sum x^2$ is the sum of squared x values
- $\sum y^2$ is the sum of squared y values

The coefficient ranges from −1 to 1, where:

- Values close to 1 indicate a strong positive correlation
- Values close to −1 indicate a strong negative correlation
- Values close to 0 indicate little to no linear correlation

1.2. Standard Library Import Ratio

The ratio of standard library imports to total imports is calculated as:

$$\text{stdLibRatio} = \frac{\text{stdLibImports.length}}{\text{allImports.length}}$$

This ratio helps us understand what portion of a project's dependencies comes from the standard library.

1.3. Average Calculations

For a collection of projects, we calculate the average of a metric using:

$$\text{Average} = \frac{\sum_{i=1}^{n} \text{metric}(\text{project}_i)}{n}$$

Where n is the number of projects and metric(project) is the value of the metric for a specific project.

1.4. LoC vs Files Ratio

We calculate the LoC vs Files ratio in two different ways:

1. **Global ratio**: Total lines of code divided by total number of files across all projects:

$$\text{Global Ratio} = \frac{\sum_{i=1}^{n} \text{LoC}_i}{\sum_{i=1}^{n} \text{Files}_i}$$

2. **Average of individual ratios**: Average of the LoC/Files ratio calculated for each project:

$$\text{Avg. Individual Ratio} = \frac{\sum_{i=1}^{n} \frac{\text{LoC}_i}{\text{Files}_i}}{n}$$

These two values can differ significantly and provide different perspectives on code organization.

1.5. Statistical Distribution Measures

For understanding the distribution of analysis times, we calculate:

- **Minimum**: The smallest value in the dataset
- **Q1 (First Quartile)**: The value below which 25% of observations are found
- **Median (Second Quartile)**: The value below which 50% of observations are found
- **Q3 (Third Quartile)**: The value below which 75% of observations are found
- **Maximum**: The largest value in the dataset

These statistics help us understand the spread and central tendency of the data without being overly influenced by outliers.

2. adactl

2.1. Fast Projects (< 0.7s)

2.1.1. Basic Statistics

Metric	Value
Number of projects	71
Average LoC	1,461.77
Average number of files	11.86
Average LoC/Files ratio (Global)	123.26
Average LoC/Files ratio (Computed by project)	196.37
Average complexity	79.92
Average analysis time (adactl)	0.087s

2.1.2. Import Statistics

Category	Count
Total	5.58
Standard Ada	2.55 (31.3%)
GNAT	0.27
System	0.07
Interface	0.25
Custom	2.44

2.1.3. Analysis Time Distribution

Metric	Time
Min	0.040s
Q1	0.050s
Median	0.067s
Q3	0.097s
Max	0.443s

2.1.4. Ada Imports by Category

Category	Count
strings	0.79
containers	0.41

Category	Count
io	0.37
memory	0.28
system interface	0.2
exceptions	0.15
numerics	0.13
other_ada	0.11
timing	0.1
tasking	0.01

2.2. Normal Projects (≥ 0.7s)

2.2.1. Basic Statistics

Metric	Value
Number of projects	47
Average LoC	1,679.15
Average number of files	18.26
Average LoC/Files ratio (Global)	91.98
Average LoC/Files ratio (Computed by project)	128.3
Average complexity	56.62
Average analysis time (adactl)	1.275s

2.2.2. Import Statistics

Category	Count
Total	6.64
Standard Ada	2.98 (28.6%)
GNAT	0.23
System	0.06
Interface	0.21
Custom	3.15

2.2.3. Analysis Time Distribution

Metric	Time
Min	0.747s
Q1	0.787s

Metric	Time
Median	0.847s
Q3	1.373s
Max	6.943s

2.2.4. Ada Imports by Category

Category	Count
strings	0.83
system interface	0.4
io	0.38
containers	0.36
memory	0.3
exceptions	0.21
timing	0.17
other_ada	0.17
numerics	0.15

2.2.5. Correlations with Analysis Time

Metric	Correlation
LoC	0.246
Number of Files	0.233
LoC / Files Ratio	0.016
Complexity	0.056
Total imports	0.107
Ada imports	0.072
GNAT imports	−0.056
Interface imports	−0.063
System imports	−0.055
Custom imports	0.127
io imports	0.007
strings imports	0.042
containers imports	0.065
timing imports	0.044
numerics imports	−0.033
tasking imports	−0.051
memory imports	0.016

Metric	Correlation
system interface imports	0.188
exceptions imports	0.054

2.3. Unique Standard/GNAT/Interface/System Imports

Present in normal projects but not in fast projects:

```
ada.calendar.conversions
ada.characters.conversions
ada.characters.wide_wide_latin_1
ada.containers.bounded_vectors
ada.containers.ordered_sets
ada.containers.synchronized_queue_interfaces
ada.containers.unbounded_synchronized_queues
ada.numerics.discrete_random
ada.strings.unbounded.text_io
ada.strings.utf_encoding
ada.strings.utf_encoding.strings
ada.strings.wide_wide_fixed
ada.strings.wide_wide_unbounded
ada.wide_wide_text_io
gnat.byte_swapping
gnat.calendar.time_io
gnat.expect
gnat.traceback.symbolic
system.parameters
```

3. gnatcheck_1cores

3.1. Fast Projects (< 0.7s)

3.1.1. Basic Statistics

Metric	Value
Number of projects	66
Average LoC	2,342.53
Average number of files	21.33
Average LoC/Files ratio (Global)	109.81
Average LoC/Files ratio (Computed by project)	229.76
Average complexity	109.65
Average analysis time (gnatcheck_1cores)	0.16s

3.1.2. Import Statistics

Category	Count
Total	6.65
Standard Ada	2.85 (27.3%)
GNAT	0.21
System	0.03
Interface	0.24
Custom	3.32

3.1.3. Analysis Time Distribution

Metric	Time
Min	0.023s
Q1	0.057s
Median	0.123s
Q3	0.203s
Max	0.640s

3.1.4. Ada Imports by Category

Category	Count
strings	0.82
io	0.39
containers	0.36
memory	0.33
system interface	0.27
exceptions	0.24
numerics	0.18
timing	0.14
other_ada	0.11

3.2. Normal Projects (≥ 0.7s)

3.2.1. Basic Statistics

Metric	Value
Number of projects	52
Average LoC	540.37

Metric	Value
Average number of files	5.62
Average LoC/Files ratio (Global)	96.23
Average LoC/Files ratio (Computed by project)	92.47
Average complexity	21.12
Average analysis time (gnatcheck_1cores)	1.166s

3.2.2. Import Statistics

Category	Count
Total	5.17
Standard Ada	2.56 (34.0%)
GNAT	0.31
System	0.12
Interface	0.23
Custom	1.96

3.2.3. Analysis Time Distribution

Metric	Time
Min	0.763s
Q1	0.867s
Median	0.880s
Q3	0.903s
Max	14.107s

3.2.4. Ada Imports by Category

Category	Count
strings	0.79
containers	0.42
io	0.35
system interface	0.29
memory	0.23
other_ada	0.17
timing	0.12
exceptions	0.1
numerics	0.08

Category	Count
tasking	0.02

3.2.5. Correlations with Analysis Time

Metric	Correlation
LoC	−0.004
Number of Files	−0.037
LoC / Files Ratio	−0.037
Complexity	−0.048
Total imports	0.031
Ada imports	0.054
GNAT imports	−0.012
Interface imports	−0.045
System imports	0.025
Custom imports	0.011
io imports	0.059
strings imports	0.064
containers imports	−0.028
timing imports	−0.026
numerics imports	−0.076
tasking imports	0.020
memory imports	0.079
system interface imports	0.093
exceptions imports	0.110

3.3. Unique Standard/GNAT/Interface/System Imports

Present in normal projects but not in fast projects:

```
ada.assertions
ada.calendar.formatting
ada.characters.conversions
ada.characters.wide_wide_latin_1
ada.containers.bounded_vectors
ada.containers.indefinite_doubly_linked_lists
ada.containers.ordered_sets
ada.strings.hash
ada.strings.utf_encoding
ada.strings.utf_encoding.strings
ada.strings.wide_wide_fixed
ada.strings.wide_wide_unbounded
```

```
ada.task_identification
ada.wide_wide_text_io
gnat.byte_swapping
gnat.expect
gnat.regpat
gnat.sockets
gnat.string_split
system.parameters
system.random_numbers
```

4. gnatcheck_32cores

4.1. Fast Projects (< 0.7s)

4.1.1. Basic Statistics

Metric	Value
Number of projects	67
Average LoC	2,299.93
Average number of files	20.79
Average LoC/Files ratio (Global)	110.62
Average LoC/Files ratio (Computed by project)	233.94
Average complexity	108
Average analysis time (gnatcheck_32cores)	0.161s

4.1.2. Import Statistics

Category	Count
Total	7.42
Standard Ada	3.43 (27.3%)
GNAT	0.27
System	0.06
Interface	0.16
Custom	3.49

4.1.3. Analysis Time Distribution

Metric	Time
Min	0.017s
Q1	0.070s
Median	0.117s

Metric	Time
Q3	0.223s
Max	0.650s

4.1.4. Ada Imports by Category

Category	Count
strings	1.1
containers	0.48
io	0.43
system interface	0.36
memory	0.33
exceptions	0.25
timing	0.21
numerics	0.18
other_ada	0.09

4.2. Normal Projects (≥ 0.7s)

4.2.1. Basic Statistics

Metric	Value
Number of projects	51
Average LoC	561
Average number of files	6.02
Average LoC/Files ratio (Global)	93.2
Average LoC/Files ratio (Computed by project)	84.28
Average complexity	21.55
Average analysis time (gnatcheck_32cores)	1.146s

4.2.2. Import Statistics

Category	Count
Total	4.14
Standard Ada	1.78 (34.1%)
GNAT	0.24
System	0.08
Interface	0.33

Category	Count
Custom	1.71

4.2.3. Analysis Time Distribution

Metric	Time
Min	0.710s
Q1	0.870s
Median	0.880s
Q3	0.900s
Max	13.257s

4.2.4. Ada Imports by Category

Category	Count
strings	0.41
io	0.29
containers	0.27
memory	0.24
other_ada	0.2
system interface	0.18
numerics	0.08
exceptions	0.08
timing	0.02
tasking	0.02

4.2.5. Correlations with Analysis Time

Metric	Correlation
LoC	−0.014
Number of Files	−0.043
LoC / Files Ratio	−0.032
Complexity	−0.054
Total imports	−0.008
Ada imports	−0.001
GNAT imports	−0.039
Interface imports	−0.004
System imports	−0.015

Metric	Correlation
Custom imports	−0.005
io imports	0.037
strings imports	−0.006
containers imports	−0.074
timing imports	−0.099
numerics imports	−0.077
tasking imports	0.023
memory imports	0.084
system interface imports	0.044
exceptions imports	0.094

4.3. Unique Standard/GNAT/Interface/System Imports

Present in normal projects but not in fast projects:

```
ada.assertions
ada.characters.conversions
ada.containers.indefinite_doubly_linked_lists
ada.containers.ordered_sets
ada.iterator_interfaces
ada.numerics
ada.numerics.generic_elementary_functions
ada.strings.utf_encoding.strings
ada.task_identification
ada.wide_wide_text_io
gnat.byte_swapping
gnat.sha256
gnat.string_split
system.parameters
```

5. Cross-Tool Cluster Comparison

Comparing 118 projects distribution between AdaControl and GNATcheck:

Category	Number of Projects	Percentage
C1 (Normal) for both tools	25	21.2%
C1 for AdaControl, C2 (Fast) for GNATcheck	22	18.6%
C2 for AdaControl, C1 for GNATcheck	27	22.9%
C2 (Fast) for both tools	44	37.3%

Table 1: Distribution of projects

GNATcheck / AdaControl	C1 (Normal)	C2 (Fast)
C1 (Normal)	25 (21.2%)	22 (18.6%)
C2 (Fast)	27 (22.9%)	44 (37.3%)

Table 2: 2x2 Contingency Table

5.1. Cluster Metric Comparison

Comparing key metrics across the four identified clusters:

5.1.1. Complexity Metrics

Cluster	Avg. LoC	Avg. Files	Avg. LoC/ Files	Avg. Com-plexity	Avg. Ada-Control Time	Avg. GNATcheck Time
C1 (Normal) for both tools	574.84	6.64	73.52	20.2	1.074s	1.437s
C1 for AdaCon-trol, C2 for GNATcheck	2,934.05	31.45	190.55	98	1.504s	0.193s
C2 for AdaCon-trol, C1 for GNATcheck	508.44	4.67	110.02	21.96	0.069s	0.915s
C2 (Fast) for both tools	2,046.77	16.27	249.36	115.48	0.098s	0.143s

Table 3: Average complexity metrics by cluster

5.1.2. Import Metrics

Cluster	Total Imports	Std. Ada	GNAT	System	Custom
C1 (Normal) for both tools	5.72	2.72	0.36	0.08	2.36
C1 for AdaControl, C2 for GNATcheck	7.68	3.27	0.09	0.05	4.05
C2 for AdaControl, C1 for GNATcheck	4.67	2.41	0.26	0.15	1.59
C2 (Fast) for both tools	6.14	2.64	0.27	0.02	2.95

Table 4: Average import metrics by cluster

5.1.3. Standard Library Categories

Cluster	io	strings	containers	timing	numerics	tasking	memory	system interface	exceptions
C1 (Normal) for both tools	0.36	0.76	0.44	0.08	0.08	0	0.32	0.32	0.12
C1 for AdaControl, C2 for GNATcheck	0.41	0.91	0.27	0.27	0.23	0	0.27	0.5	0.32
C2 for AdaControl, C1 for GNATcheck	0.33	0.81	0.41	0.15	0.07	0.04	0.15	0.26	0.07
C2 (Fast) for both tools	0.39	0.77	0.41	0.07	0.16	0	0.36	0.16	0.2

Table 5: Average standard library imports by category and cluster

5.2. C1 (Normal) vs C2 (Fast) Comparison

Comparing metrics between normal (C1) and fast (C2) projects.

All clusters may contain AdaControl and GNATcheck.

5.2.1. Complexity Metrics

Metric	C1 (Normal)	C2 (Fast)
Projects	74	93
Avg. LoC	1,252	1,810.05
Avg. Files	13.3	16.49
Avg. LoC/Files (Global)	94.15	109.74
Avg. LoC/Files (Computed by project)	121.63	195
Avg. Complexity	43.97	84.19
Avg. AdaControl Time	0.81s	0.067s
Avg. GNATcheck Time	0.819s	0.113s

Table 6: Average complexity metrics by category

5.2.2. Import Metrics

Import Type	C1 (Normal)	C2 (Fast)
Total Imports	5.92	6.08
Std. Ada	2.77	2.72
GNAT	0.24	0.23
System	0.09	0.06
Custom	2.58	2.82

Table 7: Average import metrics by category

5.2.3. Standard Library Categories

Category	C1 (Normal)	C2 (Fast)
io	0.36	0.38
strings	0.82	0.82
containers	0.38	0.38
timing	0.16	0.14
numerics	0.12	0.15
tasking	0.01	0.01
memory	0.24	0.28
system interface	0.35	0.27
exceptions	0.16	0.19

Table 8: Average standard library imports by category

5.2.4. Unique Imports Between C1 and C2

Comparing non-custom imports that are unique to C1 (Normal) vs C2 (Fast) projects:

5.2.4.1. Unique to C1 (Normal) projects

```
ada.characters.conversions
ada.characters.wide_wide_latin_1
ada.containers.bounded_vectors
ada.containers.ordered_sets
ada.strings.utf_encoding
ada.strings.utf_encoding.strings
ada.strings.wide_wide_fixed
ada.strings.wide_wide_unbounded
ada.wide_wide_text_io
gnat.byte_swapping
gnat.expect
system.parameters
```

5.2.4.2. Unique to C2 (Fast) projects

```
ada.containers.indefinite_holders
ada.containers.indefinite_ordered_multisets
ada.containers.multiway_trees
ada.numerics
ada.numerics.float_random
ada.numerics.generic_complex_types
ada.numerics.generic_elementary_functions
ada.strings.unbounded.hash
ada.tags
ada.text_io.complex_io
ada.text_io.text_streams
gnat.command_line
gnat.sha256
gnat.source_info
gnat.strings
```

5.2.5. Unique Standard/GNAT/Interface/System Imports

Comparing non-custom imports that are unique to each cluster:

5.2.5.1. Unique to C1 (Normal) for both tools

```
ada.characters.conversions
ada.characters.wide_wide_latin_1
ada.containers.bounded_vectors
ada.containers.ordered_sets
ada.strings.utf_encoding
ada.strings.utf_encoding.strings
ada.strings.wide_wide_fixed
ada.strings.wide_wide_unbounded
ada.wide_wide_text_io
gnat.byte_swapping
gnat.expect
system.parameters
```

5.2.5.2. Unique to C1 for AdaControl, C2 for GNATcheck

```
ada.calendar.conversions
ada.containers.synchronized_queue_interfaces
ada.containers.unbounded_synchronized_queues
ada.strings.unbounded.text_io
gnat.calendar.time_io
```

5.2.5.3. Unique to C2 for AdaControl, C1 for GNATcheck

```
ada.assertions
ada.calendar.formatting
ada.containers.indefinite_doubly_linked_lists
ada.strings.hash
ada.task_identification
gnat.regpat
```

```
gnat.string_split
system.random_numbers
```

5.2.5.4. Unique to C2 (Fast) for both tools

```
ada.containers.indefinite_holders
ada.containers.indefinite_ordered_multisets
ada.containers.multiway_trees
ada.numerics
ada.numerics.float_random
ada.numerics.generic_complex_types
ada.numerics.generic_elementary_functions
ada.strings.unbounded.hash
ada.tags
ada.text_io.complex_io
ada.text_io.text_streams
gnat.command_line
gnat.sha256
gnat.source_info
gnat.strings
```

5.3. List of project by distribution

5.3.1. C1 (Normal) for both tools

```
src/aaa/aaa.gpr
src/ada_pretty/gnat/ada_pretty.gpr
src/apdf/pdf_out_gnat_w_gid.gpr
src/audio_base/audio_base.gpr
src/blake2s/blake2s.gpr
src/chests/chests.gpr
src/cobs/cobs.gpr
src/dg_loada/dg_loada.gpr
src/edc_client/edc_client.gpr
src/endianness/endianness.gpr
src/epoll/epoll.gpr
src/fastpbkdf2_ada/fastpbkdf2_ada.gpr
src/getopt/getopt.gpr
src/json/json/json_pretty_print.gpr
src/lal_highlight/highlight.gpr
src/libhello/libhello.gpr
src/midi/midi.gpr
src/partord/partord.gpr
src/pbkdf2/pbkdf2.gpr
src/raspberry_bsp/raspberry_bsp.gpr
src/rsfile/rsfile.gpr
src/simh_tapes/simh_tapes.gpr
src/stopwatch/stopwatch.gpr
src/system_random/system_random.gpr
src/utf8test/utf8test.gpr
```

5.3.2. C1 for AdaControl, C2 (Fast) for GNATcheck

```
src/ada_fuse/ada_fuse.gpr
src/adabots/adabots.gpr
src/asfml/asfml.gpr
src/atomic/atomic.gpr
src/automate/automate.gpr
src/bingada/bingada.gpr
src/cbsg/cbsg.gpr
src/dashera/dashera.gpr
src/eagle_lander/eagle_lander.gpr
src/emacs_gpr_query/emacs_gpr_query.gpr
src/emojis/emojis.gpr
src/gid/gid.gpr
src/hal/hal.gpr
src/inotify/monitor.gpr
src/remoteio/remoteio.gpr
src/sdlada/build/gnat/sdlada.gpr
src/septum/septum.gpr
src/si_units/si_units.gpr
src/simple_components/tables.gpr
src/trendy_test/trendy_test.gpr
src/vaton/vaton.gpr
src/wordlist/wordlist.gpr
```

5.3.3. C2 for AdaControl, C1 for GNATcheck

```
src/ajunitgen/ajunitgen.gpr
src/b2ssum/b2ssum.gpr
src/canberra_ada/canberra_ada.gpr
src/chacha20/chacha20.gpr
src/cmd_ada/cmd_ada.gpr
src/dotenv/dotenv.gpr
src/eeprom_i2c/eeprom_i2c.gpr
src/esp_idf/esp_idf.gpr
src/get_password/get_password.gpr
src/hello/hello.gpr
src/hmac/hmac.gpr
src/loga/loga.gpr
src/minirest/minirest.gpr
src/play_2048/play_2048.gpr
src/rtmidi/rtmidi.gpr
src/saatana/saatana.gpr
src/spark_unbound/spark_unbound.gpr
src/spdx/spdx.gpr
src/tiny_text/tiny_text.gpr
src/tlsada/tlsada.gpr
src/uri_ada/uri_ada.gpr
src/uri_mime/uri_mime.gpr
src/virtapu/virtapu.gpr
src/workers/workers.gpr
src/xdg_base_dir/xdg_base_dir.gpr
```

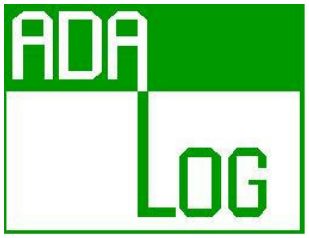
```
src/xmlada/input_sources/xmlada_input.gpr
src/xoshiro/xoshiro.gpr
```

5.3.4. C2 (Fast) for both tools

```
src/ada_lua/ada_lua.gpr
src/ada_toml/ada_toml.gpr
src/adl_middleware/adl_middleware.gpr
src/aicwl/sources/aicwl-editor.gpr
src/audio_wavefiles/audio_wavefiles.gpr
src/aunit/lib/gnat/aunit.gpr
src/bar_codes/bar_codes_gnat.gpr
src/basalt/basalt.gpr
src/brackelib/brackelib.gpr
src/c_strings/c_strings.gpr
src/dir_iterators/dir_iterators.gpr
src/evdev/evdev_info.gpr
src/ews/ews.gpr
src/excel_writer/excel_out_gnat.gpr
src/freetypeada/freetype.gpr
src/geo_coords/geo_coords.gpr
src/hangman/hangman.gpr
src/hungarian/hungarian.gpr
src/ini_files/ini_files.gpr
src/j2ada/j2ada.gpr
src/json/json/json.gpr
src/jwt/gnat/jwt.gpr
src/linenoise_ada/linenoise.gpr
src/littlefs/littlefs.gpr
src/mandelbrot_ascii/mandelbrot_ascii.gpr
src/mcp2221/mcp2221.gpr
src/parse_args/parse_args.gpr
src/powerjoular/powerjoular.gpr
src/resources/resources.gpr
src/semantic_versioning/semantic_versioning.gpr
src/simple_components/components-gnutls.gpr
src/simple_components/components-sqlite.gpr
src/simple_logging/simple_logging.gpr
src/slip/slip.gpr
src/socketcan/src/socketcan.gpr
src/sparknacl/sparknacl.gpr
src/svd2ada/svd2ada.gpr
src/tiled_code_gen/tiled_code_gen.gpr
src/toml_slicer/toml_slicer.gpr
src/trendy_terminal/trendy_terminal.gpr
src/weechat_ada/weechat_ada.gpr
src/xmlada/dom/xmlada_dom.gpr
src/xmlada/sax/xmlada_sax.gpr
src/yeison/yeison.gpr
```

C.3 Language Feature Analysis

This report details the usage of various Ada language features across the benchmarked projects. It provides the quantitative data supporting the case studies on performance factors discussed in Section 3.2.3.



Benchmark report

Language Feature Usage vs Performance (LoC <= 10,000)

2025-12-14

Université de Caen Normandie, France
Adalog SAS, SIREN 527 695 704, France

Contents

1. Overview

This report analyzes how the usage of specific Ada language features correlates with analysis time for each tool, considering only projects whose size is below a configurable LoC threshold.

2. Correlation Model

We use the Pearson correlation coefficient to measure the linear relationship between analysis time and feature usage. For each feature and each tool, we compute the correlation between:

- raw feature count (number of occurrences)
- normalized feature count (occurrences per 1k LoC)

3. Language Features Glossary

The following table summarizes each Ada language feature (“trait”) measured in this benchmark, with a short description of what is counted for each metric.

Feature	Description
Attr Access All	Number of uses of attribute ‘Access on any object.
Attr Address All	Number of uses of attribute ‘Address on any object.
Attr Unchecked Access All	Number of uses of attribute ‘Unchecked_Access on any object.
Decls Operators Overloaded	Number of declarations of overloaded operators.
Derivations Depth Protected	Maximum inheritance depth of protected types (number of derivation levels above the root).
Derivations Depth Tagged	Maximum inheritance depth of tagged types (number of derivation levels above the root).
Derivations Depth Task	Maximum inheritance depth of task types (number of derivation levels above the root).
Derivations Depth Untagged	Maximum inheritance depth of untagged types (number of derivation levels above the root).
Derivations Parents	Maximum number of parents (interfaces or base types) for any single type.
Exceptions Declared	Number of exception declarations.
Generics Decl Local	Number of locally declared generic units (generic packages or subprograms declared inside another unit).
Generics Inst Local	Number of local instantiations of generic units (inside subprograms or nested scopes).
Generics Inst Private	Number of private generic instantiations.
Generics Inst Public	Number of public generic instantiations.

Feature	Description
Generics Units All	Number of generic units (generic packages and subprograms).
Handlers Others All	Number of exception handlers using the others choice.
Handlers Others Null	Number of others exception handlers whose body is null.
Inst Unchecked Conv Addr To Access Full	Number of instantiations of Ada.Unchecked_Conversion converting System.Address to an access type (fully qualified).
Inst Unchecked Conv Addr To Access Short	Number of instantiations of Unchecked_Conversion converting System.Address to an access type (short form).
Known Exceptions Access	Number of statically known access-related exceptions (invalid pointer dereferences).
Known Exceptions Assignment	Number of statically known exceptions related to assignments (e.g., constraint errors on assignment).
Known Exceptions Index	Number of statically known index-related exceptions (out-of-range array indexing).
Known Exceptions Raise Expression	Number of statically known exceptions raised by raise expressions.
Known Exceptions Zero Divide	Number of statically known zero-divide exceptions.
Local Exception	Number of exceptions that are locally handled.
Metrics Functions Called	Number of function call sites.
Metrics Objects All	Number of object declarations or usages (rough measure of data size).
Metrics Procedures Called	Number of procedure call sites.
Metrics Statements All	Total number of executable statements.
Metrics Types Used	Number of type usages in the source (rough measure of type variety/complexity).
Named Number Declarations	Number of named number declarations (integer constants used as named numbers).
Parameter Aliasing Certain	Number of parameter aliasing situations that are certainly aliasing (definite aliases).
Parameter Aliasing Possible	Number of parameter aliasing situations that are possibly aliasing (potential aliases).
Pragmas All	Total number of pragmas.
Pragmas Nonstandard	Number of nonstandard (implementation-defined) pragmas.
Protected Objects Declared	Number of protected object declarations.

Feature	Description
Representation Clauses All	Number of representation clauses (record layout, alignment, etc.).
Statements Abort	Number of abort statements.
Statements Accept	Number of accept statements.
Statements Conditional Entry Call	Number of conditional entry call statements.
Statements Delay Relative	Number of relative delay statements (delay until a duration has elapsed).
Statements Delay Until	Number of delay until statements (delay until a specific time).
Statements Entry Call	Number of simple entry call statements.
Statements Raise All	Number of raise statements (all exceptions).
Statements Raise Standard	Number of raise statements raising predefined (standard) exceptions.
Statements Requeue	Number of requeue statements.
Statements Selective Accept	Number of selective accept statements (select).
Statements Terminate Alternative	Number of terminate alternatives in selective accept statements.
Statements Timed Entry Call	Number of timed entry call statements.
Tasks Declared	Number of task declarations.
Tasks Terminating	Number of tasks that are known to terminate (per static analysis).
Type Usage Pos On Enum	Number of uses of attribute ‘Pos on enumeration types.
Types Abstract	Number of abstract type declarations.
Types Access Subprogram	Number of access-to-subprogram type declarations.
Types Controlled	Number of controlled type declarations.
Types Derived	Number of derived type declarations.
Types Tagged With Primitives	Number of tagged types that have at least one visible primitive operation.
Types With Discriminants	Number of type declarations with discriminants.

Table 1: Language feature descriptions

4. Tool: adactl

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Metrics Objects All	0.245	0.056	100.69	187.94
Statements Terminate Alternative	0.208	0.205	0	0.13

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Tasks Terminating	0.202	−0.025	0.04	0.38
Protected Objects Declared	0.201	0.077	0	0.17
Statements Timed Entry Call	0.2	0.091	0	0.11
Statements Selective Accept	0.2	0.198	0	0.23
Parameter Aliasing Possible	0.196	0.197	0	0.04
Generics Inst Local	0.185	0.124	1	2.77
Exceptions Declared	0.179	−0.021	1.25	1.98
Metrics Types Used	0.178	−0.046	28.37	49.81
Statements Accept	0.174	0.039	0.04	0.79
Statements Delay Relative	0.171	0.025	0.24	0.51
Statements Entry Call	0.17	0.007	0.13	1.38
Statements Raise All	0.17	−0.055	6.11	11.26
Generics Decl Local	0.163	0.31	0	0.21
Types Derived	0.146	−0.057	1.58	2.23
Statements Abort	0.146	0.146	0	0.02
Statements Conditional Entry Call	0.146	0.146	0	0.02
Metrics Statements All	0.146	−0.039	341.38	342.66
Attr Access All	0.139	0.01	3.48	5.21
Metrics Procedures Called	0.137	0.041	23.99	29.17
Derivations Depth Protected	0.132	0.051	0	0.04
Types Tagged With Primitives	0.131	−0.085	3.32	4.49
Generics Inst Public	0.13	−0.045	1.37	1.85
Generics Units All	0.13	0.031	1.31	2.83
Tasks Declared	0.129	−0.04	0.01	0.09

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Pragmas All	0.123	0.005	16.86	54.57
Representation Clauses All	0.109	0.014	1.75	4.6
Metrics Functions Called	0.108	0.014	26.35	44.64
Attr Unchecked Access All	−0.093	−0.112	0.7	0.04
Derivations Parents	0.085	−0.051	0.14	0.13
Pragmas Nonstandard	0.08	0.004	4.69	11.68
Local Exception	0.068	0.01	0.01	0.06
Types Abstract	−0.062	−0.088	0.68	0.17
Generics Inst Private	−0.062	−0.123	1.37	0.32
Known Exceptions Raise Expression	−0.049	−0.06	0.07	0.02
Known Exceptions Access	−0.046	−0.046	0.01	0
Handlers Others Null	−0.043	−0.055	0.11	0.02
Type Usage Pos On Enum	0.039	−0.042	3.31	1.4
Named Number Declarations	0.038	−0.008	7.45	7.19
Attr Address All	0.037	0.021	1.14	5.38
Statements Raise Standard	−0.035	−0.058	2.32	1.77
Derivations Depth Untagged	0.03	−0.05	0.54	0.55
Types With Discriminants	−0.028	−0.113	1.49	0.7
Inst Unchecked Conv Addr To Access Full	−0.022	−0.065	0.04	0.02
Types Controlled	−0.018	−0.103	0.79	0.51
Decls Operators Overloaded	0.015	−0.04	1.14	0.81

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Parameter Aliasing Certain	−0.012	−0.012	0.1	0
Handlers Others All	0.01	−0.045	1.7	1.53
Derivations Depth Tagged	−0.01	−0.087	1.17	0.74
Types Access Subprogram	0.005	−0.074	1.01	1.15
Derivations Depth Task	0	0	0	0
Inst Unchecked Conv Addr To Access Short	0	0	0	0
Known Exceptions Assignment	0	0	0	0
Known Exceptions Index	0	0	0	0
Known Exceptions Zero Divide	0	0	0	0
Statements Delay Until	0	0	0	0
Statements Requeue	0	0	0	0

Table 2: Correlation and averages for adactl

5. Tool: gnatcheck_1cores

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Derivations Depth Untagged	−0.151	−0.022	0.77	0.25
Statements Raise Standard	−0.076	−0.057	3.35	0.52
Types Access Subprogram	−0.074	−0.059	1.77	0.17
Types Controlled	−0.069	−0.059	1.05	0.21
Metrics Functions Called	−0.068	−0.028	53.55	8.37
Attr Access All	−0.068	−0.082	6.82	0.81
Attr Unchecked Access All	−0.065	−0.045	0.76	0.04

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Metrics Types Used	−0.063	−0.045	59.26	8.54
Inst Unchecked Conv Addr To Access Full	−0.061	−0.057	0.06	0
Types Derived	−0.06	−0.045	2.98	0.38
Types Abstract	0.059	0.056	0.7	0.19
Handlers Others All	−0.058	−0.035	2.59	0.42
Statements Raise All	−0.057	−0.056	12.82	2.25
Metrics Objects All	−0.053	−0.02	215.52	33.81
Derivations Parents	−0.048	−0.052	0.24	0
Generics Units All	−0.048	−0.031	2.89	0.67
Generics Inst Private	−0.047	−0.052	1.58	0.15
Type Usage Pos On Enum	−0.045	−0.045	4.36	0.25
Derivations Depth Protected	−0.044	−0.039	0.03	0
Generics Inst Public	−0.038	−0.034	2.53	0.33
Pragmas All	−0.038	−0.007	47.95	11.48
Generics Inst Local	0.037	0.054	2.32	0.92
Protected Objects Declared	−0.035	−0.04	0.12	0
Tasks Declared	−0.034	−0.043	0.08	0
Known Exceptions Access	−0.031	−0.031	0.02	0
Types With Discriminants	−0.031	−0.021	1.44	0.85
Representation Clauses All	−0.03	0.014	4.44	0.9
Derivations Depth Tagged	0.028	0.011	1.27	0.65
Attr Address All	−0.027	−0.026	4.74	0.4
Statements Abort	−0.025	−0.025	0.02	0
Statements Conditional Entry Call	−0.025	−0.025	0.02	0

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Statements Delay Relative	−0.024	−0.052	0.61	0.02
Decls Operators Overloaded	0.021	−0.014	1.61	0.25
Generics Decl Local	−0.02	−0.013	0.14	0.02
Statements Timed Entry Call	−0.016	−0.037	0.08	0
Named Number Declarations	0.016	0.011	9.73	4.33
Tasks Terminating	−0.015	−0.023	0.29	0.04
Parameter Aliasing Possible	−0.013	−0.012	0.03	0
Known Exceptions Raise Expression	−0.013	−0.028	0.06	0.04
Metrics Procedures Called	0.01	0.045	41.67	6.23
Pragmas Nonstandard	0.009	0.031	6.98	8.1
Statements Terminate Alternative	−0.009	−0.008	0.09	0
Metrics Statements All	−0.009	−0.015	539.45	91.13
Exceptions Declared	0.008	0.004	2.33	0.54
Statements Selective Accept	−0.006	−0.006	0.17	0
Types Tagged With Primitives	0.006	0.001	5.45	1.67
Parameter Aliasing Certain	−0.005	−0.005	0.11	0
Statements Accept	−0.003	−0.025	0.59	0.02
Handlers Others Null	−0.002	0.014	0.08	0.08
Local Exception	−0.002	0.027	0.03	0.04
Statements Entry Call	−0.001	0	1.02	0.13
Derivations Depth Task	0	0	0	0

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Inst Unchecked Conv Addr To Access Short	0	0	0	0
Known Exceptions Assignment	0	0	0	0
Known Exceptions Index	0	0	0	0
Known Exceptions Zero Divide	0	0	0	0
Statements Delay Until	0	0	0	0
Statements Requeue	0	0	0	0

Table 3: Correlation and averages for gnatcheck_1cores

6. Tool: gnatcheck_32cores

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Derivations Depth Untagged	−0.135	−0.006	0.73	0.29
Statements Raise Standard	−0.073	−0.034	3.34	0.47
Metrics Functions Called	−0.073	−0.064	54.22	6.59
Metrics Types Used	−0.072	−0.074	58.49	8.55
Local Exception	−0.067	−0.06	0.06	0
Statements Raise All	−0.066	−0.058	12.97	1.84
Attr Access All	−0.066	−0.044	6.28	1.39
Types Controlled	−0.061	−0.038	1.04	0.2
Types Access Subprogram	−0.058	0.003	1.6	0.37
Attr Unchecked Access All	−0.055	−0.032	0.7	0.1
Metrics Objects All	−0.055	−0.049	208.99	38.82
Types Derived	−0.052	−0.021	2.82	0.55
Handlers Others Null	−0.051	−0.045	0.12	0.02

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Known Exceptions Raise Expression	−0.05	−0.044	0.09	0
Handlers Others All	−0.047	−0.033	2.4	0.63
Generics Units All	−0.046	0.016	2.81	0.75
Types With Discriminants	−0.045	−0.018	1.55	0.69
Derivations Depth Protected	−0.041	−0.037	0.03	0
Generics Inst Public	−0.039	−0.011	2.42	0.43
Pragmas All	−0.036	0.01	46.69	12.43
Types Tagged With Primitives	−0.033	−0.029	6.1	0.75
Known Exceptions Access	−0.032	−0.032	0.01	0
Types Abstract	0.032	0.023	0.78	0.08
Derivations Parents	−0.032	0.005	0.21	0.04
Pragmas Nonstandard	−0.031	0.021	8.1	6.65
Tasks Declared	−0.031	−0.044	0.07	0
Generics Inst Private	−0.031	−0.017	1.52	0.2
Generics Inst Local	0.03	0.024	2.42	0.76
Inst Unchecked Conv Addr To Access Full	−0.028	−0.001	0.04	0.02
Protected Objects Declared	−0.027	−0.038	0.09	0.04
Derivations Depth Tagged	0.026	0.03	1.27	0.65
Attr Address All	−0.023	−0.015	4.61	0.49
Statements Abort	−0.022	−0.022	0.01	0
Statements Conditional Entry Call	−0.022	−0.022	0.01	0
Generics Decl Local	−0.021	−0.015	0.13	0.02
Metrics Statements All	−0.02	−0.035	521.13	106.41

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Representation Clauses All	−0.019	0.03	4.1	1.27
Named Number Declarations	0.015	0.008	10.07	3.76
Statements Delay Relative	−0.015	0.001	0.43	0.24
Metrics Procedures Called	0.013	0.038	40.72	6.78
Type Usage Pos On Enum	−0.011	0.005	3.42	1.41
Statements Entry Call	0.01	0.008	0.18	1.22
Decls Operators Overloaded	0.009	−0.037	1.66	0.16
Statements Accept	0.008	−0.019	0.1	0.65
Parameter Aliasing Possible	−0.006	−0.005	0.01	0.02
Statements Timed Entry Call	−0.005	−0.033	0.03	0.06
Statements Selective Accept	0.005	0.006	0.04	0.16
Parameter Aliasing Certain	−0.004	−0.004	0.1	0
Tasks Terminating	−0.004	−0.019	0.13	0.24
Exceptions Declared	0.003	0.022	2.18	0.71
Statements Terminate Alternative	0.002	0.003	0.03	0.08
Derivations Depth Task	0	0	0	0
Inst Unchecked Conv Addr To Access Short	0	0	0	0
Known Exceptions Assignment	0	0	0	0
Known Exceptions Index	0	0	0	0
Known Exceptions Zero Divide	0	0	0	0

Feature	Corr(count, time)	Corr(density, time)	Avg count (fast)	Avg count (slow)
Statements Delay Until	0	0	0	0
Statements Re-queue	0	0	0	0

Table 4: Correlation and averages for gnatcheck_32cores

Declaration of Generative AI and AI-assisted technologies in the writing process

During the drafting of the manuscript, the author employed POE[1], a wrapper across multiple chat assistance tool, primarily for the purpose of refining the writing style. The tool was utilized solely for enhancing the coherence, formality, and academic tone of the manuscript. After employing POE, the author thoroughly reviewed and edited the content as needed, ensuring accuracy and clarity in conveying the research findings. The author takes full responsibility for the content of the manuscript, recognizing that while the tool contributed to stylistic improvements, ultimate accountability rests with the human author for the substantive and scholarly aspects of the work.

[1] POE has been used to encompass the following LLMs: Assistant of Poe.com, Claude.ai, Chat GPT-4

Acronyms

API Application Programming Interface. 20, 47, 48, 50

ASIS Ada Semantic Interface Specification. xxi, xxvii, xxix, 20–22, 42, 47, 62, 64, 86, 92, 102

AST Abstract Syntax Tree. xviii, xx–xxii, xxv, xxvii, 2, 8, 9, 13, 14, 18, 20, 21, 29, 34–37, 39, 41, 47, 51, 52, 57, 58, 62, 63, 67, 76, 78, 81, 82, 86, 89–91, 93, 94, 96, 97, 99, 101, 102

BNF Backus-Naur Form. 20

CFG Control Flow Graph. xx, 11–13, 99

CG Call Graph. xx, 10, 12, 14, 35, 37, 51

CI/CD Continuous Integration / Continuous Development. 17, 92, 102

CPG Code Property Graph. xx, 13–18, 21, 23, 26, 29, 35, 94

CPU Central Processing Unit. 63

CV Coefficient of Variation. 71, 72

GDBMS Graph DataBase Management System. 5, 23, 29–31, 49, 52

GPU Graphics Processing Unit. 64

GQL Graph Query Language. 25

GUI Graphical User Interface. 60

HTTP HyperText Transfer Protocol. 48, 50, 52

IDE Integrated Development Environment. 93, 102

JSON JavaScript Object Notation. 48, 50

LoC Lines of Code. 42, 43, 48, 59, 72, 73, 76–82

PDG Program Dependency Graph. xx, 12, 13, 16, 99

SCA Static Code Analysis. xxxvii, 8, 42

SCG Semantic Code Graph. 15–17, 27

Table of contents

Optimized data structure and query system for static code analysis

Abstract

Static code analysis encompasses various techniques for improving software quality and security. In this research, we focus exclusively on one important aspect: the verification of coding rules. Conventional approaches for coding rule verification face challenges in efficiently analyzing large, complex codebases. We thus explore the potential of graph databases to enhance the performance of this specific static analysis task.

We propose a graph-based methodology that represents source code as rich property graphs, enabling intuitive modeling of syntax, semantics, and behavior specifically for coding rule verification. We parse the codebase and populate it into a graph database. Then, we evaluate coding rules through graph traversals expressed in the Cypher query language, converting traditional rule checks into optimized graph patterns.

We implemented this approach in a prototype tool, entitled Cogralys, for Ada and evaluated it on real-world benchmarks. Our experiments demonstrate significant runtime improvements in coding rule verification: Cogralys completes analyses 6.3 times faster than AdaControl and 17.6 times faster than GNATcheck. For specific rule categories, we achieved even greater improvements– up to 195 times faster for local rules compared to traditional analyzers. These results confirm graph databases' capacity to accelerate coding rule verification through optimized data structures and parallel query processing.

However, overheads introduced by database population should be considered. We found the technique is best suited for sizable, frequently analyzed code. While showing significant promise for coding rule verification, more research is needed to address language support, integration with developers' workflows, and queries for more complex rules.

Overall, in this thesis we deliver a practical graph-based framework for coding rule verification while presenting the advantages, trade-offs and future opportunities of leveraging graph technologies for efficient, scalable verification of coding standards.

Keywords: ada language, graph databases, neo4j, pattern matching, scalability, static code analysis, coding rule verification

---

GREYC

6 Boulevard du Maréchal Juin – Bâtiment Sciences 3 – CS 14032 – 14032 CAEN cedex 5 – France

Structures de données et système de requêtes optimisés pour l'analyse statique de code

Résumé

L'analyse statique de code englobe diverses techniques pour améliorer la qualité et la sécurité des logiciels. Dans notre recherche, nous nous concentrons exclusivement sur un aspect important : la vérification des règles de codage. Nous avons identifié que les approches conventionnelles pour la vérification des règles de codage peinent à analyser efficacement des bases de code volumineuses et complexes. Nous explorons le potentiel des bases de données orientées graphe pour améliorer les performances de cette tâche spécifique d'analyse statique.

Nous proposons une méthodologie basée sur les graphes pour représenter le code source sous forme de graphe de propriétés, permettant une modélisation intuitive de la syntaxe, de la sémantique et du comportement spécifiquement pour la vérification des règles de codage. Nous analysons la base de code et l'intégrons dans une base de données orientée graphe. Nous évaluons ensuite les règles de codage par des traversées de graphe exprimées en langage de requête Cypher, convertissant les vérifications traditionnelles en motifs de graphe optimisés.

Nous avons implémenté cette approche par l'intermédiaire d'un prototype, appelé Cogralys, pour le langage Ada et l'avons évaluée sur des benchmarks du monde réel. Nos expériences démontrent des améliorations significatives en temps d'exécution pour la vérification des règles de codage : Cogralys effectue les analyses 6,3 fois plus rapidement qu'AdaControl et 17,6 fois plus rapidement que GNATcheck. Pour certaines catégories de règles, nous avons obtenu des améliorations encore plus importantes – jusqu'à 195 fois plus rapide pour les règles locales par rapport aux analyseurs traditionnels. Ces résultats confirment la capacité des bases de données graphe à accélérer la vérification des règles de codage grâce à des structures de données optimisées et à un traitement parallèle des requêtes.

Cependant, nous reconnaissons que les surcharges introduites par la population de la base de données sont à prendre en compte. Nous avons constaté que la technique est mieux adaptée pour du code volumineux et fréquemment analysé. Bien que prometteuse pour la vérification des règles de codage, nous identifions que des recherches supplémentaires sont nécessaires pour traiter la prise en charge d'autres langages, l'intégration dans les flux de développement et les requêtes pour des règles plus complexes.

Globalement, dans cette thèse nous proposons un cadre basé sur les graphes pour la vérification des règles de codage tout en présentant les avantages, les inconvénients et les opportunités futures de l'utilisation des technologies graphes pour une vérification efficace et évolutive des standards de codage.

Mots clés : langage ada, bases de données orientées graphe, neo4j, filtrage par motif, passage à l'échelle, analyse statique de code, vérification de règles de codage