INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Statische Prüfung von Prolog Quellcode

Enum Cohrs

Bachelorarbeit

Beginn der Arbeit:	2017-12-11
Abgabe der Arbeit:	2018-03-12
Erstgutachter:	Prof. Dr. Michael Leuschel
Zweitgutachter:	Prof. Dr. Michael Schöttner

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 2018-03-12

Enum Cohrs

Inhaltsverzeichnis

1. Einleitung

Mit zunehmender Komplexität wird Software oft fehleranfällig, schwer wartbar und unübersichtlich. Zur guten Praxis in der Softwareentwicklung gehört es daher, Programme einfach zu halten, und schwierige Stellen im Blick zu behalten, zu vereinfachen und zu dokumentieren. Um den Komplexitätsgrad von Software zu messen, wurde in den vergangenen Jahrzehnten eine Vielzahl von Metriken entwickelt, von denen einige in dieser Arbeit vorgestellt werden.

Kaum verzichtbar ist es bei großen Projekten auch, in Kommentaren und übersichtlichen Nachschlagewerken die Funktionsweise und Eigenschaften der Module und Unterprogramme nachvollziehen zu können. Damit hierbei keine doppelte Arbeit anfällt, sind in einigen Programmiersprachen bereits Werkzeuge verbreitet, die Quelltextkommentare automatisch verarbeiten und daraus HTML-Seiten oder PDF-Dokumente erzeugen. Beispiele dafür sind JavaDoc, Doxygen und Haddock.

Infolog[4] ist ein bislang von Jens Bendisposto, Michael Leuschel und Sebastian Krings entwickeltes Programm, das den Prolog-Quelltext von ProB analysiert, einige Komplexitätsmaße bereits anwendet und einige typische Probleme automatisch erkennen kann. Die Ergebnisse können dabei in CSV- und EDN-Dateien exportiert, sowie im Webbrowser visuell dargestellt werden.

Im Rahmen dieser Arbeit habe ich Infolog um zusätzliche Komplexitätsmaße erweitert und eine automatische Dokumentationsgenerierung hinzugefügt, die mit der Komplexitäts- und Problemanalyse zusammenarbeitet. Neben einfacher Beschreibung von Modulen, Prädikaten und ihren Argumenten unterstützt Infolog dabei auch Rechtfertigungen über die Verwendung von Metaprädikaten und die Beschreibung und Prüfung von Seiteneffekten. Zudem habe ich die Makefile so angepasst, dass neben ProB auch beliebige andere Prolog-Projekte analysiert werden können, die keine nichttrivialen Buildanweisungen benötigen.

2. Komplexitätsmaße

Will man die Komplexität eines Programms gering, und somit die Fehleranfälligkeit möglichst niedrig halten, ist es notwendig, sich einen Überblick zu verschaffen, welche Teile des Quelltextes denn die größte Komplexität aufweisen. Hierzu lassen sich feste Metriken definieren.

Archer [3] definiert für gute Komplexitätsmetriken folgende Charakteristiken:

1. „The measure should be robust. The calculation of the measure is repeatable and the result is insensitive to minor changes in environment, tool, or observer. The measure is precise, and the process of collecting the data for the measure is objective.

2. The measure should suggest a norm, scale, and bounds. There is a scale upon which we can make a comparison of two measures of the same type, and so conclude which of the two measures is more desirable. For example, there is a realistic lower bound, such as zero for number of errors.

3. The measure should be meaningful. The measure relates to the product, and there should be a rationale for collecting data for the measure.“

Verschiedene Metriken orientieren sich auch an unterschiedlichen Auffassungen davon, was unter „Komplexität“ zu verstehen ist [13], und beantworten verschiedene Fragen:

1. Wie viel Entwicklungsaufwand steckte in dem Programm(teil)? [7]

2. Wie schwierig ist es, den Quelltext zu verstehen? [16]

3. Wie schnell können bei der Wartung Fehler passieren? [13]

4. Wie aufwändig ist es, Fehler zu finden und zu beheben? [14]

Viele heute allgemein übliche Komplexitätsmaße sind für objektorientierte Programmierung ausgelegt. So wird z.B. die Anzahl von Klassen in einem Modul oder Namespace gemessen, die Anzahl der öffentlichen, überladenen oder aller Methoden [9], die Anzahl der Kind- oder Elternklassen [5], oder auch die Anzahl der Attribute [9].

Da Prolog keine objektorientierte, sondern eine logische Programmiersprache ist, sind all diese Maße nicht anwendbar. Es bleiben einige Maße, die sehr allgemein formuliert und damit paradigmenübergreifend sind, jedoch lassen sich auch neue finden, die auf Prologs Eigenschaften ausgelegt sind.

Im Folgenden werde ich auf einige in Prolog anwendbare Metriken eingehen.

2.1. Anzahl der Codezeilen

Ein sehr einfaches und allgemeines Maß, das sich an geradezu jedes durch Quelltext beschriebene Programm anlegen lässt, ist die Anzahl der Quelltextzeilen (Lines of Code, LOC). Dabei handelt es sich um ein reines Volumenmaß: Die LOC können nur Auskunft darüber geben, wie viel Quelltext vorhanden ist, nicht wie schwierig dieser zu verstehen ist. Sie lassen sich jedoch flexibel an verschieden große Programmeinheiten anlegen: an das ganze Programm, an ein Modul, an ein Prädikat oder auch an nur eine Klausel.

In vielen Programmiersprachen, darunter auch Prolog, ist es nicht syntaktisch vorgeschrieben, wo ein Zeilenumbruch gesetzt wird. Die Entscheidung darüber treffen Entwickler*innen meist willkürlich. Nicht selten hängt sie mit der bevorzugten Breite des Editorfensters zusammen, oder mit Ästhetik. Eine kleine Änderung im Style Guide kann die LOC verdoppeln oder halbieren ohne an der tatsächlichen Komplexität etwas zu ändern. Schon daher erfüllt die Metrik das Kriterium der Robustheit nicht.

Ein Beispiel für ein Prädikat, bei dem die Anzahl der Codezeilen irreführend ist, ist das lange, aber äußerst simple Prädikat `infolog_help/0` aus Infolog [4].

```
1  infolog_help :-
2    nl,
3    print('INFOLOG_ENTRY: analyze("/path/to/prob/src/prob_tcltk.pl", "name of
         meta_user_pred_cache")'),nl,
4    print('INFOLOG_ENTRY: dot_gen_dep(Module), dot_gen_dep(Module,0)'),nl,
5    print('INFOLOG_ENTRY: rem(Module) - Equivalence classes of predicates in
         module'),nl,
6    print('INFOLOG_ENTRY: compute_cycles - compute cyclic module dependencies'),
         nl,
7    print('INFOLOG_ENTRY: cycle_ids(Module,Len,Path) - iterative deepening search
         for module cycles'),nl,
8    print('INFOLOG_ENTRY: compute_call_cycles(From,Call) - compute cyclic call
         dependencies'),nl,
9    print('INFOLOG_ENTRY: pred_links(Module,Predicate) - compute cyclic call
         dependencies'),nl,
10   print('INFOLOG_ENTRY: pu(Module) - print required use_module directives'),nl,
11   print('INFOLOG_ENTRY: print_uia - print useless use_module directives'),nl,
12   print('INFOLOG_ENTRY: print_reexports - print predicates re-exported'),nl,
13   print('INFOLOG_ENTRY: dca - dead code analysis'),nl,
14   print('INFOLOG_ENTRY: complexity - clause complexity analysis'),nl,
15   print('INFOLOG_ENTRY: lint - find problems'),nl,
16   print('INFOLOG_ENTRY: print_meta_calls(Module)'),nl,
17   print('INFOLOG_ENTRY: print_undocumented_sideeffects'), nl,
18   print('INFOLOG_ENTRY: sccs'),nl,
19   nl,nl.
```

Dagegen ist `uncovered_call/6` deutlich kürzer, benötigt aber mehr Zeit zum Verständnis:

```
1 uncovered_call(FromModule,FromQ,ToModule,Call,L1,L2) :- calling(FromModule,
      FromQ,ToModule,Call,L1,L2),
2     (always_defined(Call) -> fail
3      ; ToModule=built_in -> fail % we assume SICStus only assigns built_in if
           it exists
4      ; is_defined(ToModule,Call)
5      -> fail, % comment in to only detect calls without definition
6         \+ check_imported(ToModule,Call,FromModule) % it is defined but not
              imported
7      ; true % it is not defined
8      ).
```

Da Infolog bereits vorher Start- und Endzeilennummern der Klauseln speicherte, war es einfach, die LOC-Angabe zum Webinterface hinzuzufügen.

2.2. Verschachtelungstiefe

Eine Metrik, die sich nicht allein auf Volumen stützt, sondern Kontrollstrukturen einbezieht, ist die Verschachtelungstiefe [17, 15] (Nested Block Depth, NBD). Die Idee dahinter ist, dass stark verschachtelte Kontrollstrukturen schwieriger nachvollziehbar sind, als mehrere flachere Kontrollstrukturen hintereinander.

In imperativen Programmiersprachen sind zumeist Schleifen, sowie If- und Case-Verzweigungen entscheidend für die NBD. In Infolog wird die Verschachtelungstiefe bei folgenden Strukturen erhöht:

- Bei Bedingungen: `(->)/2`, `if/3`
- bei Alternativen: `(;)/2`
- bei Negationen: `(\+)/1`
- und in Metaprädikaten, z.B. `catch/3`, `findall/3`, `setof/3`

Diese haben zu If- und Case-Verzweigungen sowie zu Schleifen die syntaktische Ähnlichkeit, dass auch sie meistens zu Blöcken eingeklammert und eingerückt werden. Ein entscheidender Unterschied besteht jedoch: If- und Case-Bedingungen und Schleifen sind in imperativen Sprachen meistens die einzigen Stellen, an denen sich der Programmfluss verzweigt, in Prolog kann sich der Programmfluss jedoch in jedem Prädikataufruf und jeder Unifikation verzweigen. Was in anderen Sprachen eine Schleife, und damit eingerückt wäre, kann in Prolog eine „fail loop" mit NBD=0 sein. Gleichzeitig zeigt die Analyse des ProB-Quelltextes [8] mit Infolog, dass die meisten Prädikate, die einen hohen NBD-Wert zugewiesen bekommen haben, sich wie ein Switch-Case-Block lesen und leicht verständlich sind.

Die Verschachtelungstiefe wurde in Infolog bereits vorher erhoben und auch im Webinterface dargestellt. Ich habe hieran nichts verändert.

2.3. Variablen

Das Zählen von Variablen ist eine Metrik, die in logischen Programmen ebenso anwendbar ist wie in imperativen. Mehr Variablen sind ein Zeichen dafür, dass mehr Informationen berücksichtigt werden müssen, also auch mehr Faktoren Einfluss auf den Programmfluss nehmen können. Mehr Variablen erhöhen auch den Aufwand, der zum Verständnis des Programm(teil)s notwendig ist, da für jede Variable ermittelt werden muss, was ihr Inhalt bedeutet und welche Rolle sie einnimmt.

In Prolog kommt dazu, dass mit mehr Variablen auch mehr Unifikationen kommen, die fehlschlagen können und Backtracking auslösen. Fehlschlagen können jedoch auch Prädikataufrufe mit wenig oder keinen Variablen, ebenso ist ein stark verzweigter Programmfluss auch mit wenig Variablen möglich. Die Zahl der Variablen ist also nur als Hinweis zu interpretieren, aber nicht als alleiniges Maß.

Um die Anzahl der Variablen in einer Klausel zu zählen, habe ich `analyze_clause_complexity/5` erweitert. Die paarweise verschiedenen Variablen werden dabei mit `term_variables/2` extrahiert.

2.4. Unifikationen

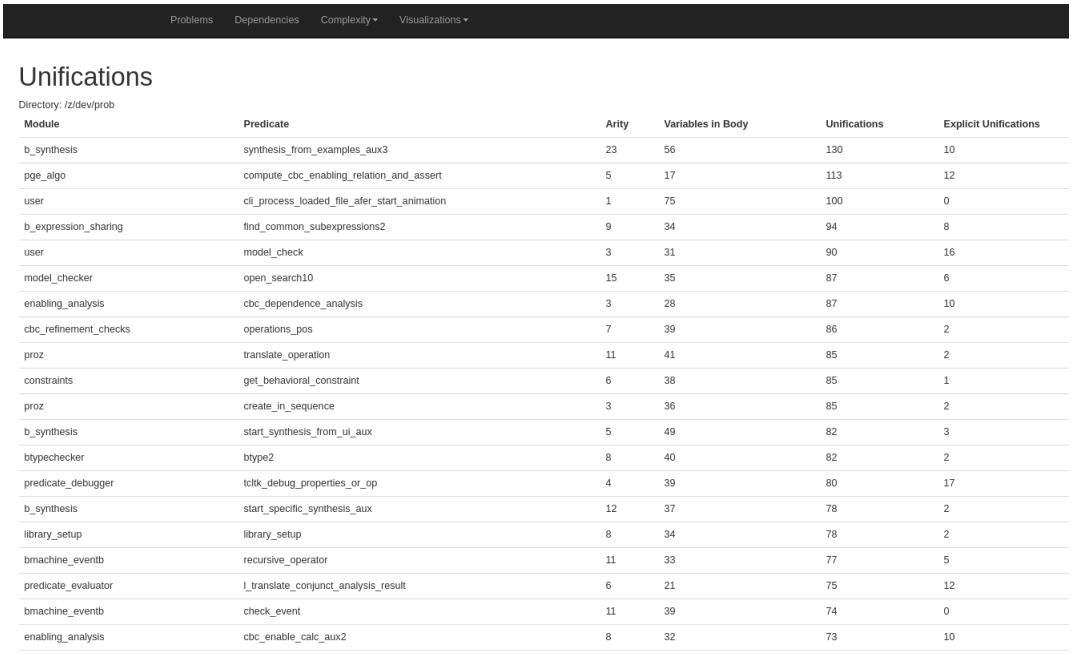
Statt Variablen lassen sich auch die Unifikationen zählen. Hiermit sind nicht nur explizite Unifikationen mit dem Operator `(=)/2` gemeint, sondern auch solche, die implizit vonstattengehen, wenn Variablen als Parameter in Prädikataufrufen verwendet werden. Dadurch fließt in die Zahl der Unifikationen nicht nur die Anzahl der Variablen indirekt ein, sondern auch die Anzahl und Arität der Aufrufe hat einen Einfluss, sofern als Argumente Variablen übergeben werden.

Beispielsweise enthält folgende Klausel aus ProB [8] 11 Unifikationen, davon zwei explizit:

```
1 b_compile2(Exp,Parameters,LS,S,NExpr,Eval,FullyKnown,WF) :- var(Exp), !,
2     add_internal_error('Variable Expression: ',b_compile2(Exp,Parameters,LS,S,
          NExpr,Eval,FullyKnown,WF)),
3     NExpr=Exp, FullyKnown=false.
```

Sicherlich lässt sich streiten, ob bei **var**(Exp) tatsächlich eine Unifikation vorliegt – jedoch lässt sich nicht allgemein und verlässlich prüfen, welche Argumente unifizierenden Charakter haben und welche nicht, daher wertet meine Zählung alle als solche.

Zwar ging beim ProB-Quelltext [8] der erste Platz bezüglich der Unifikationenanzahl an ein Prädikat, das lediglich Beispieldaten definiert und sonst ganz simpel ist (`synthesis_from_examples_aux3`),



Problems Dependencies Complexity Visualizations

Unifications

Directory: /z/dev/prob

Module	Predicate	Arity	Variables in Body	Unifications	Explicit Unifications
b_synthesis	synthesis_from_examples_aux3	23	56	130	10
pge_algo	compute_cbc_enabling_relation_and_assert	5	17	113	12
user	cli_process_loaded_file_afer_start_animation	1	75	100	0
b_expression_sharing	find_common_subexpressions2	9	34	94	8
user	model_check	3	31	90	16
model_checker	open_search10	15	35	87	6
enabling_analysis	cbc_dependence_analysis	3	28	87	10
cbc_refinement_checks	operations_pos	7	39	86	2
proz	translate_operation	11	41	85	2
constraints	get_behavioral_constraint	6	38	85	1
proz	create_in_sequence	3	36	85	2
b_synthesis	start_synthesis_from_ui_aux	5	49	82	3
btypechecker	btype2	8	40	82	2
predicate_debugger	tcltk_debug_properties_or_op	4	39	80	17
b_synthesis	start_specific_synthesis_aux	12	37	78	2
library_setup	library_setup	8	34	78	2
bmachine_eventb	recursive_operator	11	33	77	5
predicate_evaluator	l_translate_conjunct_analysis_result	6	21	75	12
bmachine_eventb	check_event	11	39	74	0
enabling_analysis	cbc_enable_calc_aux2	8	32	73	10

Abbildung 2.1.: Anzahl der Variablen, Unifikationen und expliziten Unifikationen in der Weboberfläche

doch bereits der zweite Platz ist ein Musterbeispiel für ein schwer nachvollziehbares Prädikat (`compute_cbc_enabling_relation_and_assert/5`), und auch die folgenden Plätze sind Treffer (`find_common_subexpressions2/9`, `model_check/3` etc.)

Zur Zählung der Unifikationen und der expliziten Unifikationen habe ich `analyze_clause_complexity/5` und `body_complexity/3` erweitert. Im Webinterface wurde dazu eine neue Tabellenseite „Unifications“ angelegt.

2.5. Calls in Body

Ein anderes Maß ist die Zählung der Aufrufe (Calls in Body, CIB) in einer Klausel. Ähnlich wie bei der Anzahl der Codezeilen handelt es sich hier um ein reines Volumenmaß, das lediglich den Programmieraufwand schätzen kann. Wie schwierig der Code zu verstehen oder zu warten ist, lässt sich daraus nicht ableiten. Im Gegensatz zur LOC-Metrik hängt das Ergebnis hier jedoch nicht von willkürlichen ästhetischen Entscheidungen oder von Style Guides ab, und ist daher robuster.

Die Zahl der Aufrufe wurde bereits von Infolog bereits vorher erhoben. Ich habe daran nichts geändert.

2.6. Ausgehende Aufrufkanten

Eine ähnliche Idee wie Calls in Body verfolgt die Zählung der paarweise verschiedenen, aufgerufenen Prädikate. In einem gerichteten Graphen, der Prädikate als Knoten und Aufrufe zwischen Prädikaten als Kanten darstellt, entspricht dies gerade der Anzahl ausgehender Kanten (Outgoing Edges, OE).

Dadurch, dass mehrmalige Aufrufe eines Prädikats nicht zu einem höheren Wert führen, ist OE nicht geeignet, den Programmieraufwand zu messen. Die Schwierigkeit, nachzuvollziehen, was in einer Klausel passiert, steigt jedoch in der Regel eher mit mehr verschiedenen Prädikaten als mit wiederholten Aufrufen des immer selben, und wird durch OE daher besser approximiert als durch CIB.

Beispielsweise hat das im Abschnitt „Anzahl der Codezeilen“ aufgeführte Prädikat `infolog_help/0` einen CIB-Wert von 35, aber einen OE-Wert von 2. Das ebenfalls dort aufgeführte Prädikat `uncovered_call/6` hat nur einen CIB-Wert von 8, aber einen OE-Wert von 6. Dies spiegelt wieder, dass `infolog_help/0` länger ist und mehr Schreibarbeit bedeutete, `uncovered_call/6` aber schwieriger zu verstehen ist.

Erhoben wird die Anzahl der ausgehenden Aufrufkanten durch das neue Prädikat `pred_incalls_outcalls/4`. Dieses zählt die von `analyze_body/5` erhobenen `calling/6`-Fakten unter Zusammenfassung gleicher Zielknoten. Im Webinterface wurde dazu eine neue Tabellenseite „Calls“ angelegt.

2.7. Eingehende Aufrufkanten

Wie ausgehende Kanten lassen sich auch eingehende Aufrufkanten zählen (Incoming Edges, IE). Für ein gegebenes Prädikat besagt diese Zahl, von wie vielen anderen Prädikaten es aufgerufen wird. Die Deutung dieser unterscheidet sich von den bisherigen Metriken: Hier lässt sich weder Programmieraufwand noch Verständnisschwierigkeit ablesen. Vielmehr sagen die IE etwas darüber aus, wie viele andere Prädikate indirekt betroffen sind, wenn in diesem Prädikat ein Fehler vorliegt oder sich die Semantik ändert. Ebenso ist ein hoher IE-Wert ein Indikator dafür, dass das Prädikat dringend gut dokumentiert und getestet werden sollte, da sich ein Fehler darin viral auf das gesamte Programm auswirkt.

Auch die Anzahl der eingehenden Aufrufkanten wird in `pred_incalls_outcalls/4` durch Zählung der `calling/6`-Fakten ermittelt.

2.8. Dynamische Prädikate

Die Anzahl dynamischer Prädikate (Dynamic Predicates, DP) bemisst, wie viele Informationen in einem Modul im Hintergrund gespeichert werden. Dies ist zumeist erst dann notwendig,

Modul	Prädikat	Incoming edges	Outgoing edges
error_manager	add_internal_error/2	424	3
error_manager	add_error/3	356	1
debug	debug_println/2	295	2
bsyntaxtree	conjunct_predicates/2	287	5
preferences	get_preference/2	267	5
module_information	module_info/2	242	6
bsyntaxtree	get_texpr_expr/2	239	3
bsyntaxtree	get_texpr_type/2	197	3
preferences	preference/2	189	5
bsyntaxtree	get_texpr_id/2	175	3

Abbildung 2.2.: Die 10 ProB-Prädikate mit den höchsten IE-Werten.

wenn ein Sachverhalt oder Algorithmus kompliziert genug ist, dass die Daten nicht mehr sinnvoll in Argumenten gehalten werden können.

Die Ergebnisse für den ProB-Quelltext (Abbildung 2.3) bestätigen die Annahme, dass nur wenige Module besonders viele dynamische Prädikate verwenden.

Modul	dynamische Prädikate
bmachine	66
sap	31
state_space	29
plugins	26
refinement_checker	19
haskell_csp_analyzer	17
b_machine_hierarchy	16
self_check	15
haskell_csp	15
eval_strings	15

Abbildung 2.3.: Die 10 ProB-Module mit den meisten dynamischen Prädikaten

Die DP-Zahl wird in `module_predicate_stats/4` durch Zählung der `is_dynamic/2`-Fakten ermittelt und im Webinterface auf der Seite „Predicates per module“ aufgeführt.

2.9. Zyklomatische Komplexität

Die zyklomatische Komplexität bemisst die Anzahl der möglichen linear unabhängigen Pfade durch ein Programm [12]. Zur Berechnung dieser schlägt McCabe [12] vor, einen Programmflussgraphen aufzubauen und die „cyclomatic number“ des Graphen wie folgt zu berechnen:

$$v(G) = e - n + 2p \tag{2.1}$$

Als einfachere Alternative zum Aufstellen des Programmflussgraphen führt McCabe in [12] außerdem folgende Vereinfachung auf:

$$v(G) = \pi + 1 \tag{2.2}$$

wobei π die Anzahl der Verzweigungen ist.

Moores greift in [14] beide Varianten auf und schlägt für Prolog folgende Vereinfachungen vor:

- „a single node, n, replaces all instantiations of the predicate (such that, a call by any predicate with the same name is represented as a call from a single node with that name);
- a predicate in the body of a clause is represented as an arc, e, from the clause head node to the clause body node (where one arc represents all calls between predicates of the same name);
- the number of conditionals contained within a program is equated to the number of unique predicate names, P“

In [14] und [13] kommt Moores durch Anwendung bei 80 Prolog-Programmen und Vergleich der Anzahl gemeldeter Softwarefehler zu dem Schluss, dass die zweite Berechnungsweise besser geeignet sei, die Fehleranfälligkeit zu messen.

Ich habe mich aus zwei Gründen dagegen entschieden, zyklomatische Komplexität in Infolog einzuführen. Erstens ergibt die theoretische Definition über den Aufrufgraph nur für ganze Programme Sinn, die sich dadurch miteinander vergleichen ließen. ProB ist jedoch im Wesentlichen ein großes Programm und das Ziel der Komplexitätsanalyse ist es, verschiedene Programmteile miteinander zu vergleichen.

Zweitens lässt sich eine Zählung der Prädikate zwar auf einzelne Module herunterbrechen, damit ist allerdings keine neue Information gewonnen, da die Anzahl der Prädikate in einem Modul schon längst in Infolog erhoben wird.

2.10. Halstead-Metriken

Halstead [7] schlägt unabhängig von Programmiersprachen vor, die folgenden vier Größen zu erheben und daraus eine Reihe abgeleiteter Metriken zu errechnen [2, 10]:

- „n_1: Number of distinct operators.
- n_2: Number of distinct operands.
- N_1: Total number of occurrences of operators.

- N_2: Total number of occurrences of operands.“

Al Qutaish und Abran [2] und Lister [10] bemängeln, dass bereits die Unterscheidung zwischen Operanden und Operatoren nicht klar definiert ist. Die genaue Unterscheidung ist daher implementierungsabhängig und Archers Robustheitskriterium ist nicht erfüllt. Da Halsteads andere Maße von den obigen vieren abgeleitet sind, gilt dasselbe für sie. Ahmad [1] stellte verschiedene Zählweisen für Prolog gegenüber und fand heraus, dass die Ergebnisse sich nur geringfügig unterscheiden, empfiehlt aber in der Zusammenfassung, eigene Prädikate als Operatoren zu zählen und Datensektionen mitzubetrachten.

Die abgeleiteten Maße werden laut [2] wie folgt berechnet:

- Die Länge N über $N = N_1 + N_2$,
- das Vokabular n über $n = n_1 + n_2$,
- das Volumen V über $V = N \cdot \log_2 n$,
- der Programmebenenschätzer $\hat{\text{L}}$ über $\hat{L} = \frac{2}{n_1} \cdot \frac{n_2}{N_2}$,
- die Schwierigkeit D über $D = \frac{1}{L}$,
- der intelligente Gehalt I über $I = \hat{L} \cdot V$,
- der Programmieraufwand E über $E = \frac{V}{L}$,
- die Programmierzeit T über $T = \frac{E}{S}$ in Sekunden mit $S = 18$.

Abseits des Programmebenenschätzers $\hat{\text{L}}$ gibt es laut [10, 2] noch die Formel für die Programmebene $L = \frac{V^*}{V}$. Nach [10] ist V^* das potenzielle Volumen eines Algorithmus und gegeben durch „the volume of the minimal program required to express it“. Da dieses in der Praxis nicht bekannt ist, arbeitet die Implementierung in Infolog mit dem Schätzer $\hat{\text{L}}$. Al Qutaish und Abran [2] bemängeln außerdem zurecht, dass aus $L = \frac{V^*}{V}$ und $I = L \cdot V$ folgen würde $I = \frac{V^*}{V} \cdot V = V^*$, was der Beschreibung der Metriken als „intelligent content“ und „potential volume“ nicht gerecht wird.

Dennoch hat sich gezeigt, dass der Programmieraufwand E besser mit der tatsächlichen Programmierzeit, der Fehlerzahl, und dem Debugaufwand korreliert als die zyklomatische Komplexität [14]. Daher habe ich eine Zählung von n_1, n_2, N_1 und N_2 in `analyze_clause_complexity/5` und `body_complexity/3` eingebaut. Im Webinterface werden in einem eigenen Reiter daraus Länge, Vokabular, Programmebenenschätzer, Schwierigkeit, intelligenter Gehalt, Programmieraufwand und Programmierzeit berechnet und aufgelistet.

Problems Dependencies Complexity Visualizations

Halstead

Directory: /z/dev/prob

Module	Predicate	Operator tokens	Operand tokens	Distinct operators	Distinct operands	Length	Vocabulary	Volume	Level estimator	Difficulty	Intelligent Content	Programming Effort	Programming Time
logging	synthesis_log_aux/1	10	959	5	57	969	62	5769.62	0.02	41.38	139.43	238742.74	13263.49 s
enabling_analysis	cbc_quick_cfg_analysis/3	31	506	17	67	537	84	3432.67	0.02	63.38	54.16	217545.74	12085.87 s
enabling_analysis	cbc_dependence_analysis/3	47	395	18	71	442	89	2862.27	0.02	49.50	57.82	141682.57	7871.25 s
latex_processor	process_dot_command/3	45	200	31	53	245	84	1566.12	0.02	57.69	27.15	90356.29	5019.79 s
user	model_check/3	87	201	28	76	288	104	1929.73	0.03	36.73	52.54	70873.60	3937.42 s
logging	synthesis_log_aux/1	5	475	4	47	480	51	2722.76	0.05	19.83	137.28	54001.49	3000.08 s
predicate_debugger	tcltk_debug_properties_or_op/4	73	165	32	82	238	114	1626.23	0.03	32.00	50.82	52039.29	2891.07 s
logging	synthesis_log_aux/1	11	316	8	45	327	53	1873.03	0.04	27.57	67.95	51630.48	2868.36 s
pge_algo	compute_cbc_enabling_relation_and_assert/5	44	161	15	25	205	40	1091.00	0.02	46.73	23.35	50983.05	2832.39 s
translate	explain_event_step/3	18	257	8	30	275	38	1443.18	0.03	33.29	43.35	48043.93	2669.11 s
translate	explain_event_step/3	12	339	5	32	351	37	1828.52	0.04	25.76	70.99	47098.19	2616.57 s
b_state_model_check	cbc_model_check/4	58	101	39	46	159	85	1019.09	0.02	42.32	24.08	43127.16	2395.95 s
user	cli_process_loaded_file_afer_start_animation/1	76	363	15	218	439	233	3452.38	0.08	12.47	276.95	43036.49	2390.92 s
debugging_calls	remove_debugging_calls/4	31	175	17	41	206	58	1206.74	0.03	35.62	33.88	42983.08	2387.95 s
user	cli_execute_aux/6	58	112	27	35	170	62	1012.21	0.02	42.38	23.89	42892.54	2382.92 s
haskell_csp	reset_for_selfcheck/0	46	491	6	136	537	142	3839.41	0.09	10.77	356.37	41364.78	2298.04 s

Abbildung 2.4.: Halstead-Metriken auf ProB [8] angewandt

3. Dokumentationssystem

Computerprogramme werden mit wachsendem Umfang und Alter schwieriger zu verstehen. Dazu trägt auch bei, dass mit der Zeit immer wieder einige Menschen zu anderen Projekten wechseln und neue dazu kommen, die ebenfalls am Quelltext mitarbeiten, oder mitarbeiten wollen. Um zu gewährleisten, dass ein Einstieg jederzeit möglich ist, ist es eine gängige Praxis in der Softwareentwicklung, den Quelltext auf zweierlei Weise zu dokumentieren.

Einerseits ist es hilfreich, wenn der Quelltext selbst reichlich mit Kommentaren versehen ist, die beschreiben, wie Programmteile funktionieren, da Code und Beschreibung so direkt beieinanderstehen. Andererseits bietet eine externe Dokumentation den Vorteil, besser durchsuch-, indizier- und verknüpfbar zu sein, und außerdem strukturiertere Darstellung zu ermöglichen, als dies mit reinem Text mitten in Quellcode ohne weiteres möglich ist.

In den letzten Jahren wurden in einigen Programmiersprachen Werkzeuge verbreitet, die beide Ansätze kombinieren, ohne den Entwickler*innen dabei doppelten Aufwand zu verursachen. JavaDoc, Doxygen, Haddock und andere extrahieren aus dem Quelltext besonders gekennzeichnete Dokumentationskommentare und erzeugen daraus Webseiten oder Druckfassungen.

Im Rahmen dieser Bachelorarbeit habe ich ein solches System für Prolog unter dem Namen InfologDoc implementiert und mit anderen Komponenten von Infolog verknüpft.

3.1. Kommentartypen

Die meisten Dokumentationssysteme dieser Art berücksichtigen nur Kommentare, die besonders dafür ausgezeichnet sind. JavaDoc [6] verlangt beispielsweise Blockkommentare, die mit einem zusätzlichen Stern eingeleitet werden:

```
1 /** Dies ist ein JavaDoc-Kommentar */
2 /* Dieser wird hingegen ignoriert */
```

In ähnlicher Weise verlangt Haddock [11] ein Pipe-Symbol am Anfang des Kommentars:

```
1 -- | Ein Haddock-Kommentar
2
3 -- Kein Haddock-Kommentar
```

Bei InfologDoc habe ich mich zunächst dagegen entschieden, nur besonders ausgezeichnete Kommentare zu berücksichtigen, stattdessen wird der erstbeste Kommentar verwendet, der

direkt vor einem Prädikat zu finden ist. Dies hat den Grund, dass Kommentare in ProB bislang ohnehin rar gesäht sind und die erzeugte Dokumentation daher zumindest die Kommentare, die InfologDoc finden konnte, auch enthalten sollte.

In der Voraussicht, dass sich dieser Zustand mit der Zeit ändern könnte, unterscheidet InfologDoc intern bereits mehrere Arten von Kommentaren:

- Einzeiler: Ein alleinstehender mit dem Prozentzeichen eingeleiteter Zeilenkommentar.
- Mehrzeiler: Mehrere aufeinander folgende, jeweils mit dem Prozentzeichen eingeleitete Zeilenkommentare.
- Haken: Mehrere aufeinander folgende, jeweils mit dem Prozentzeichen eingeleitete Zeilenkommentare, wobei der erste abweichend mit zwei Prozentzeichen beginnt.
- Normaler Block: Ein regulärer Blockkommentar, der durch /* eingeleitet und durch */ beendet wird.
- Gesternter Block: Ein Blockkommentar, bei der jede weitere Zeile mit einem Stern beginn.
- JavaDoc: Ein Blockkommentar in JavaDoc-Syntax.

Beispiele für die Kommentartypen sehen wie folgt aus:

```
1  % foo/0 ist mit einem Einzeiler versehen.
2  foo.
3
4  % foo/1 hingegen hat
5  % einen Mehrzeiler.
6  foo(_).
7
8  %%
9  % foo/2 beginnt gar
10 % mit einem Hakenkommentar.
11 foo(_,_).
12
13 /* goo/0 hat einen
14    ganz normalen Blockkommentar. */
15 goo.
16
17 /* goo/1 hat auch einen Blockkommentar
18  * aber mit Sternchen */
19 goo(_).
20
21 /** goo/2 hat schliesslich
22  *  einen JavaDoc-Kommentar */
23 goo(_,_).
```

3.2. Tags

Wie in JavaDoc können die Kommentare mit Tags versehen sein. Jede Tagangabe muss in einer eigenen Zeile stehen und mit einem @-Zeichen beginnen. Direkt darauf folgt der Name des Tags, dann ein Leerzeichen und der Inhalt. Der Inhalt eines Tags kann sich auf mehrere Zeilen erstrecken, das Tag endet erst dann, wenn der Kommentar geschlossen wird oder die nächste Tag-Angabe folgt. Alle angegebenen Tags werden in die generierte Dokumentation aufgenommen.

Beispiele für sinnvolle Tags sind `@author`, `@date` oder `@version`. Da InfologDoc die verfügbaren Tags nicht auf einen festen Satz beschränkt, sondern alle annimmt, die es findet, ist hier viel möglich. Drei Tagnamen haben jedoch eine besondere Semantik: `@param`, `@justify` und `@sideeffect`.

Auf `@param` folgt erst der Name des Parameters, danach eine Beschreibung. Sind `@param`-Tags angegeben, so sollten sie die Reihenfolge einhalten, die auch bei Aufrufen erwartet wird, da InfologDoc daraus automatisch eine Parameterliste generiert.

```
1 %%
2 % Verbruezemt mit dem Bazong einen Barambaz und gibt das Parakumt zurueck.
3 % @author Marvin Cohrs
4 % @param Bazong das verwendete Bazong
5 % @param Barambaz der Barambaz, der verbruezemt werden soll
6 % @param Parakumt das fertige Parakumt
7 foo(_,_,_).
```

foo/3 (Bazong, Barambaz, Parakumt)

Verbruezemt mit dem Bazong einen Barambaz und gibt das Parakumt zurueck.

author Marvin Cohrs

param Bazong das verwendete Bazong

param Barambaz der Barambaz, der verbruezemt werden soll

param Parakumt das fertige Parakumt

Mit `@sideeffect` lässt sich dokumentieren, dass das Prädikat einen Seiteneffekt hat, seine Ausführung also einen Zustand verändert. Nach dem Tag folgt erst der Name des Seiteneffektes, dann eine genauere Beschreibung. Die folgende Liste enthält Beispiele für Seiteneffekte, es lassen sich jedoch auch beliebig weitere einführen.

- „print“ haben alle Prädikate, die in irgendeinem Auswertungszweig Ausgaben auf die Konsole oder in einen Stream tätigen;

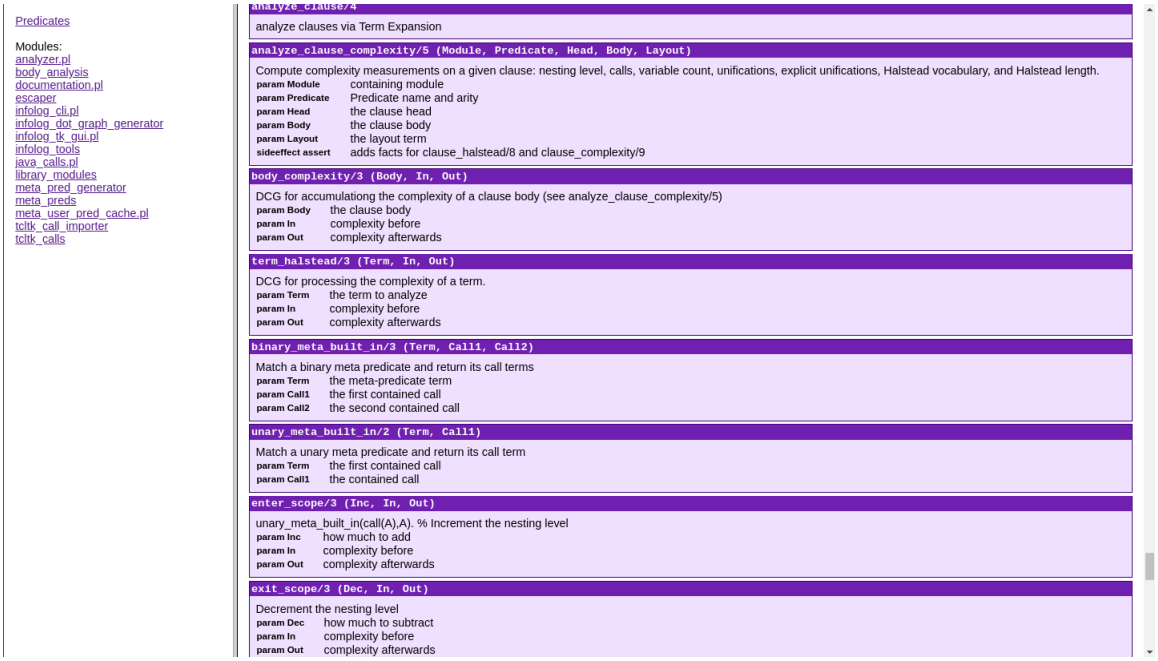


Abbildung 3.1.: Generierte HTML-Seite für den Prolog-Analyzer

- „open“ oder „close“ haben alle Prädikate, die einen Dateistream öffnen bzw. schließen;
- „assert“ oder „retract“ haben alle Prädikate, die Fakten über dynamische Prädikate hinzufügen oder löschen.

Schließlich lassen sich mit `@justify` Rechtfertigungen anbringen, warum ein bestimmtes rechen- oder speicherintensives Prädikat verwendet wird. Insbesondere vorgesehen ist dies für Aufrufe von `findall`, `bagof` und `setof`. Nach `@justify` folgt erst der Name des Prädikats, dann die Rechtfertigung.

```
1 %%
2 % nice predicate
3 % much wow
4 % @sideeffect print Barks to the console.
5 % @justify findall Required for finding all friends!
6 doge(X) :- print(bark(X)), nl, findall(Y,friend(X,Y),_).
```

3.3. Arbeitsweise

InfologDoc verarbeitet zunächst die gegebenen Kommandozeilenoptionen und ruft dann den DirectoryTraverser auf. Dieser delegiert den Import der Problemdatenbank an den ProblemImporter, der die CSV-Datei einliest und die Problemeinträge an den IndexStore meldet. Der

DirectoryTraverser durchsucht danach rekursiv das Eingabeverzeichnis nach Prolog-Dateien und übergibt diese an den Parser.

Der Parser erhebt nicht den Anspruch, die gesamte Prolog-Syntax detailliert zu prüfen, sondern teilt den Quelltext lediglich in „Chunks“ ein. Ein „Chunk“ ist dabei entweder ein Zeilenkommentar, ein Blockkommentar, ein Klauselkopf oder ein Klauselkörper. Klauselköpfe und Klauselkörper können zusammengehören, müssen dies aber nicht. Fakten (Klauseln ohne Ziele) haben einen Kopf aber keinen Körper. Deklarationen der Form „:– use_module(...).“ haben einen Körper, aber keinen Kopf.

Die Ausgaben des Parsers werden zusammengeführt durch den Joiner. Dieser erkennt, welche Klauselköpfe und -körper zusammengehören, und wo ein neues Prädikat beginnt. Den erkannten Prädikaten werden ihre Dokumentationskommentare zugeordnet, und Zeilenkommentare dabei vereinigt. JoinedComment parset die angegebenen Tags.

Danach wird der ausgewählte Generator aufgerufen. Es gibt zwei Generatoren: HtmlGenerator und LatexGenerator. Der Generator erzeugt ein Ausgabedokument, das für jedes Prädikat die gewonnenen Informationen in einem eigenen Abschnitt zusammenfasst.

Die gefundenen Prädikate werden dann durch den DirectoryTraverser in den IndexStore eingetragen. Nachdem alle Dateien verarbeitet wurden, wird der IndexGenerator aufgerufen. Sofern HTML als Ausgabeformat gewählt wurde, erzeugt dieser ein Modulverzeichnis, ein Prädikatenverzeichnis und eine Startseite.

Anschließend werden durch den PrologExporter Fakten über die Seiteneffektangaben und Metaprädikat-Rechtfertigungen als Prolog-Quelltext exportiert.

3.4. Interopabilität mit analyzer.pl

InfologDoc arbeitet auf zweierlei Art mit dem Prolog-Analyzer zusammen.

3.4.1. Seiteneffekt-Prüfung und Metaprädikat-Rechtfertigungen

Die oben aufgeführten Tags `@sideeffect` und `@justify` werden nicht nur in die generierten HTML- und LaTeX-Dokumente übernommen, sondern auch beim Export mit `--export-prolog` in die Datei `documentation.pl` geschrieben.

Bei der nächsten Erzeugung der Problemdatenbank durch `analyzer.pl` prüft `find_all_sideeffects/0`, welche Prädikate direkt oder transitiv von Seiteneffekten betroffen sind. Wird dabei für ein Prädikat ein Seiteneffekt gefunden, der nicht durch `@sideeffect` dokumentiert ist, wird eine Warnung generiert. Diese soll die Entwickler*innen dazu anhalten, den Seiteneffekt durch ein `@sideeffect`-Tag zu beschreiben, sodass bei der Verwendung des Prädikats Überraschungen vermieden werden.

Ausgangspunkte der transitiven Suche nach Seiteneffekten sind dabei `print/1`, `nl/0`, `format/2`, `format/3`, `open/3`, `close/1`, `assert/1`, `asserta/1`, `assertz/1`, `retract/1` und `retractall/1`,

sowie alle Prädikate bei denen ein `@sideeffect`-Tag bereits angegeben ist.

Desweiteren wird bei der Erzeugung der Problemdatenbank auch geprüft, ob für jede Verwendung der Metaprädikate `findall/3`, `findall/4`, `bagof/3` und `setof/3` eine Rechtfertigungsklausel gegeben ist. Liegt für ein Prädikat keine vor, wird ebenfalls eine Warnung generiert. Im Gegensatz zur Seiteneffekt-Prüfung wird dies jedoch nicht transitiv verfolgt, sondern nur direkte Aufrufe berücksichtigt.

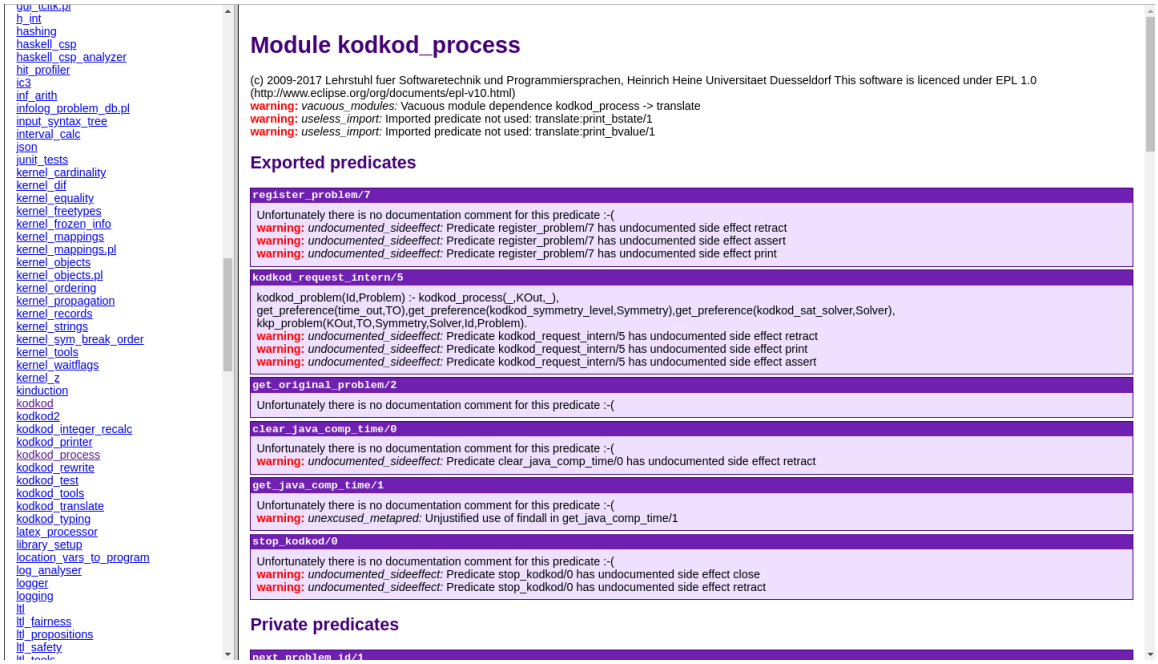


Abbildung 3.2.: ProB glänzt bislang mehr durch Warnungen als durch Beschreibungen.

3.4.2. Import der Problemdatenbank

Damit Problemmeldungen nicht nur unübersichtlich in einer großen zentralen Tabelle landen, sondern auch bei den betroffenen Modulen und Prädikaten direkt zu finden sind, importiert InfologDoc alle durch den Prolog-Analyzer gefundenen Probleme aus der Datei `infolog_problems.csv`. Die Probleme werden den Modulen und Prädikaten zugeordnet und in den jeweiligen Abschnitten der generierten HTML- oder LaTeX-Dateien mitaufgelistet.

3.5. Verwendung

Die Erzeugung eines vollständigen Ausgabedokuments wird dadurch nichttrivial, dass InfologDoc sowohl Informationen für den Prolog-Analyzer exportiert, gleichzeitig aber auch dessen Resultate benötigt. Der normale Ablauf ist daher dreischrittig:

1. Aufruf von InfologDoc mit `--no-docs` und `--export-prolog`, um die Datei `documentation.pl` zu erzeugen.
2. Aufruf des Prolog-Analyzers zur Erzeugung von `infolog_problems.csv` unter Verwendung der `documentation.pl`.
3. Aufruf von InfologDoc mit `--html` bzw. `--latex` und `--problems-csv` zur Erzeugung der gewünschten Ausgabedokumente unter Verwendung der `infolog_problems.csv`.

Zur Vereinfachung dessen gibt es in der Makefile die Targets `prolog-analyzer/documentation.pl`, `docs` und `infologdoc.pdf`. `prolog-analyzer/documentation.pl` wird automatisch als Abhängigkeit für `infolog_problems.csv` und `infolog.edn` erzeugt. Die anderen beiden Targets müssen selbst aufgerufen werden, wenn die Erzeugung der HTML-Seite unter `resources/public/docs/` bzw. des PDF-Dokuments `infologdoc.pdf` gerade gewünscht ist.

Zur genaueren Steuerung des Verhaltens von InfologDoc werden folgende Kommandozeilen-Optionen akzeptiert:

`--html` Erzeuge HTML-Seiten (Standard).

`--latex` Erzeuge LaTeX-Dokumente.

`--no-docs` Erzeuge keine Ausgabedokumente.

`--preamble` Erzeuge in den LaTeX-Dokumenten eine Präambel (Standard).

`--no-preamble` Erzeuge in den LaTeX-Dokumenten keine Präambel.

`--privates-too` Liste bei Modulen auch nicht-exportierte Prädikate auf (Standard).

`--exports-only` Liste bei Modulen nur exportierte Prädikate auf.

`--start-page` Erzeuge im Ausgabeverzeichnis eine Datei `index.html` mit einem Frameset (Standard).

`--no-start-page` Erzeuge keine Datei `index.html`.

`--modules-index` Erzeuge im Ausgabeverzeichnis eine Datei `modules.html` mit Links zu den einzelnen Modulen (Standard).

`--no-modules-index` Erzeuge keine Datei `modules.html`.

`--predicate-index` Erzeuge im Ausgabeverzeichnis eine Datei `predicates.html` mit Links zu den einzelnen Prädikaten (Standard).

`--no-predicate-index` Erzeuge keine Datei `predicates.html`.

`--out` ***out-dir*** Setze *out-dir* als Ausgabeverzeichnis (Standard: `.`).

`--css` ***css-path*** Setze *css-path* als zu verwendenden Stylesheet (Standard: keiner).

`--export-prolog` ***filename*** Exportiere die Seiteneffekt- und Rechtfertigungsfakten nach *filename* (Standard: kein Export).

`--problems-csv` ***filename*** Importiere die Problemdatenbank aus *filename* (Standard: kein Import).

4. Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurden folgende Komplexitätsmaße neu implementiert, die alle auch über das Webinterface betrachtbar sind:

- Anzahl der Variablen (Variable Count, VC),
- Anzahl der Unifikationen (Unification Count, UC),
- Anzahl der expliziten Unifikationen (Explicit Unification Count, EUC),
- Ausgehende Aufrufkanten (Outgoing Edges, OE),
- Eingehende Aufrufkanten (Incoming Edges, IE),
- Anzahl dynamischer Prädikate (Dynamic Predicates, DP),
- sowie die Halstead-Metriken
  - Länge N,
  - Vokabular n,
  - Volumen V,
  - Programmstufe L,
  - Schwierigkeit D,
  - Intelligenter Gehalt I,
  - Programmieraufwand E und
  - Programmierzeit T.

Das entwickelte Dokumentationssystem InfologDoc hilft dabei, einen Überblick über Zweck und Arbeitsweise aller Module und Prädikate in einem Projekt zu behalten, sowie die erwarteten Parameter zu beschreiben. In Zusammenarbeit mit dem Prolog-Analyzer wird dabei geprüft, ob alle Seiteneffekte der Prädikate in der Dokumentation aufgezählt und beschrieben wurden, sodass spätere Mitentwickler*innen , die den Code aufrufen oder ändern, nicht von unerwarteten Zustandsänderungen oder Ausgaben überrascht werden. Der Rechtfertigungszwang für `findall`, `setof` und `bagof` soll dazu führen, dass diese Prädikate nicht leichtfertig

an Stellen benutzt werden, an denen auch eine einfachere Lösung möglich wäre. Schließlich ermöglicht InfologDoc auch, dass Problemmeldungen geordnet bei den Prädikaten und Modulen aufgeführt werden, auf die sie sich beziehen.

Die aus dieser Bachelorarbeit hervorgegangenen Quelltexte liegen sowohl auf der beigefügten CD bereit, als auch auf GitHub unter der URL `https://github.com/asyndeton/infolog`. Ein Pull Request an das ursprüngliche Quellrepositorium wurde gestellt.

A. Systembeschreibung

A.1. analyzer.pl

In `analyzer.pl` wurden von mir folgende nicht-dynamische Prädikate hinzugefügt:

`module_predicate_stats/4` (Module,Dynamics,Public,Exported)
: Zählt die als dynamisch, öffentlich oder exportiert gekennzeichneten Prädikate in einem Modul.

`pred_incalls_outcalls/4` (Module,Predicate,InCount,OutCount)
: Zählt die eingehenden und ausgehenden Aufrufkanten eines Prädikats.

`find_all_sideeffects/0` (-)
: Findet alle transitiv vererbten Seiteneffekte, indem es die Aufrufpfade verfolgt. Die Ergebnisse werden in `has_sideeffect/2` festgehalten.

`find_all_side_effects/1` (MaxIter)
: Generiert neue Fakten für `has_sideeffect/2`, indem es die in vorherigen Iterationen begonnenen Aufrufpfade weiterverfolgt.

`has_undocumented_sideeffect/2` (Predicate,SideEffect)
: Gleicht `has_sideeffect/2` und `has_documented_sideeffect/3` ab und ermittelt dadurch, welche Seiteneffekte noch nicht dokumentiert sind.

`print_undocumented_sideeffects/0` (-)
: Sucht mittels `find_all_sideeffects/0` alle Seiteneffekte und gibt die undokumentierten davon auf der Konsole aus.

`demand_justification_for/2` (Predicate,Alias)
: Hier wird festgelegt, für welche Metaprädikate eine Rechtfertigung mittels einer „@justify“-Angabe verlangt werden soll. Das zweite Argument gibt dabei an, welcher Name dabei in der „@justify“-Angabe für das Prädikat verwendet wird. Dieser kann, muss aber nicht mit dem Prolognamen des Prädikats übereinstimmen.

`has_unexcused_metapred_call/2` (Predicate,MetaPredicate)
: Sucht Prädikate, die unter `demand_justification_for/2` aufgeführte Metaprädikate verwenden, aber bei denen keine Rechtfertigung mit einer „@justify“-Angabe vorliegt.

`add_eunif/2` (In,Out)
: Erhöht bei der Komplexitätsanalyse durch `body_complexity/3` die Zähler für Unifikationen und explizite Unifikationen.

`inc_operator_occ/2` (In,Out)
: Erhöht bei der Komplexitätsanalyse durch `body_complexity/3` den Zähler für Operatorvorkommen gemäß Halstead [7].

`ins_operator/3` (Operator,In,Out)
: Fügt bei der Komplexitätsanalyse durch `body_complexity/3` der Operatorenmenge einen neuen Operator hinzu.

`inc_operand_occ/2` (In,Out)
: Erhöht bei der Komplexitätsanalyse durch `body_complexity/3` den Zähler für Operandenvorkommen gemäß Halstead [7].

`ins_atom/3` (Atom,In,Out)
: Fügt bei der Komplexitätsanalyse durch `body_complexity/3` der Atommenge ein neues Atom hinzu.

Dynamische Prädikate:

`clause_halstead/8` (Module,Predicate,StartLine,EndLine,OperatorOcc,OperandOcc,DistinctOperators,DistinctOperands)
: Speichert die Ergebnisse der Klauseln bezüglich Halsteads Grundmetriken.

`has_side_effect/2` (Predicate,SideEffect)
: Speichert die (transitiven) Seiteneffekte von Prädikaten.

Automatisch erzeugte Prädikate:

`is_documented/1` (Predicate)
: Speichert, dass ein Prädikat mit einem Dokumentationskommentar versehen ist.

`has_metapred_excuse/3` (Predicate,MetaPredicate,Excuse)
: Speichert, dass für die Nutzung eines unter `demand_justification_for/2` aufgeführten Metaprädikats eine Rechtfertigung gegeben wurde.

`has_documented_sideeffect/3` (Predicate,SideEffect,Comment)
: Speichert, dass bei einem Prädikat ein Seiteneffekt dokumentiert wurde.

A.2. Dokumentationssystem

Das Dokumentationssystem verwendet folgende Klassen, Schnittstellen und Enumerationstypen:

`BlockCommentChunk` (Klasse) > CommentChunk
: Repräsentiert einen Codeabschnitt, der einen Blockkommentar enthält.

`Chunk` (Klasse)
: Repräsentiert einen von Parser ausgegebenen Codeabschnitt.

`ChunkType` (Enum)
: Unterscheidet verschiedene Arten von Codeabschnitten.

`ClauseBodyChunk` (Klasse) > Chunk
: Repräsentiert einen Codeabschnitt, der den Körper einer Klausel enthält.

`ClauseHeadChunk` (Klasse) > Chunk
: Repräsentiert einen Codeabschnitt, der den Kopf einer Klausel enthält.

`CommentChunk` (Klasse) > Chunk
: Repräsentiert einen Codeabschnitt, der einen Kommentar enthält.

`CommentStyle` (Enum)
: Unterscheidet verschiedene Arten von Kommentaren.

`DirectoryTraverser` (Enum)
: Durchforstet ein Verzeichnis und ruft für die gefundenen Prolog-Dateien den Parser und den Generator auf.

`Generator` (Interface)
: Schnittstelle für Klassen, die Ausgabedokumente generieren, etwa HtmlGenerator und LatexGenerator.

`GeneratorFactory` (Interface)
: Schnittstelle für Factories, die Generator- und IndexGenerator-Instanzen erzeugen.

`HtmlGenerator` (Klasse) implementiertGenerator
: Ein Generator, der HTML-Dokumente erzeugt.

`IndexGenerator` (Interface)
: Schnittstelle für Klassen, die Modul- und Prädikatenverzeichnisse sowie eine Startseite generieren.

`IndexStore` (Klasse)
: Speichert während der Analyse alle gefundenen Module, Prädikate und importierten Probleme.

`JoinedComment` (Klasse)
: Repräsentiert einen gefundenen und aufbereiteten Dokumentationskommentar zu einem Prädikat oder einem Modul.

`Joiner` (Klasse)
: Führt Klauselköpfe und -körper zusammen und ordnet Dokumentationskommentare den Prädikaten zu.

`LatexGenerator` (Klasse) implementiertGenerator
: Ein Generator, der LaTeX-Dokumente erzeugt.

`LineCommentChunk` (Klasse) > CommentChunk
: Repräsentiert einen Codeabschnitt, der einen Zeilenkommentar enthält.

`Parser` (Klasse)
: Liest eine Prolog-Datei ein und spaltet sie auf in Kommentare, Klauselköpfe und Klauselkörper.

`Predicate` (Klasse)
: Repräsentiert ein Prädikat samt Dokumentationskommentar.

`Preferences` (Klasse)
: Verwaltet die gesetzten Einstellungen und Pfade.

`ProblemImporter` (Klasse)
: Importiert die von `analyzer.pl` gefundenen Probleme aus `infolog_problems.csv`.

`PrologExporter` (Klasse)
: Exportiert die Fakten über `is_documented/1`, `has_metapred_excuse/3` und `has_documented_sideeffect/3` in `documentation.pl`.

Details sind den JavaDoc-Dokumenten auf der CD zu entnehmen.

B. Verzeichnisstruktur

Auf der CD liegen die Dateien wie folgt angeordnet:

```
/
  thesis.pdf
    (Diese Arbeit)
  thesis/
    thesis.tex
      (Hauptquelldatei der thesis.pdf)
    chapters/...
      (Kapiteldateien)
    images/...
      (Grafiken)
  autogenerated/
    javadoc/...
      (JavaDoc-Ausgabe für InfologDoc)
    html/...
      (InfologDoc-Ausgabe für Prolog-Analyzer als HTML)
    infologdoc.pdf
      (InfologDoc-Ausgabe für Prolog-Analyzer als PDF)
  infolog/
    prolog-analyzer/...
      (die Quelldateien des Prolog-Analyzers)
    resources/...
      (das Ausgabeverzeichnis für das Webinterface)
    src/...
      (die Quelldateien des Webinterfaces)
    analyzers/
      doc/
        (hier liegen die Quelltexte für InfologDoc)
      doc.jar
        (InfologDoc, fertig gebaut)
      indy.jar
```

```
    (Einrückungsanalyse)
Makefile
  (Steuerungsdatei für GNU Make)
README.md
  (Kurzbeschreibung des Projekts)
project.clj
  (Projektdatei für das Webinterface)
lein
  (Leiningen-Hilfsskript)
```

C. Referenzen

C.1. Literatur

[1] Aftab Ahmad. "Software Science Analysis of PROLOG". In: *Journal of King Saud University - Computer and Information Sciences* 6 (1992), S. 1–16. ISSN: 1319-1578. URL: http://masder.kfnl.gov.sa/handle/123456789/14069.

[2] Rafa E. Al-Qutaish und Alain Abran. "An Analysis of the Design and Definitions of Halstead's Metrics". In: *Proceedings of the 15th International Workshop on Software Measurement (IWSM'2005)*. Montréal, Canada: Shaker Verlag, 2005, S. 337–352.

[3] Clark Archer und Michael Stinson. *Object-Oriented Software Measures*. Techn. Ber. CMU/SEI-95-TR-002. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1995. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12329.

[4] Jens Bendisposto, Michael Leuschel und Sebastian Krings. *Infolog*. 2013–2017. URL: https://github.com/bendisposto/infolog.

[5] S. R. Chidamber und C. F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), S. 476–493. ISSN: 0098-5589. DOI: 10.1109/32.295895.

[6] Oracle Corporation. *JavaDoc-Referenz*. 1993, 2018. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.

[7] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[8] Michael Leuschel u. a. *ProB*. 2009–2017. URL: https://www3.hhu.de/stups/prob/index.php/Main_Page.

[9] W. Li und S. Henry. "Maintenance metrics for the object oriented paradigm". In: *Proceedings First International Software Metrics Symposium*. 1993, S. 52–60. DOI: 10.1109/METRIC.1993.263801.

[10] A. M. Lister. "Software Science–The Emperor's New Clothes?" In: *The Australian Computer Journal* 14.2 (1982), S. 66–70. ISSN: 0004-8917.

[11] Simon Marlow und David Waern. *Haddock*. 2002–2018. URL: http://hackage.haskell.org/package/haddock.

[12] T. J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), S. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[13] Trevor T. Moores. "Applying Complexity Measures to Rule-based Prolog Programs". In: *Journal of Systems and Software* 44.1 (Dez. 1998), S. 45–52. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(98)10042-0. URL: http://dx.doi.org/10.1016/S0164-1212(98)10042-0.

[14] Trevor T. Moores. "Cyclomatic complexity as a measure of the structure of knowledge-based/expert systems". In: *Software Quality and Productivity: Theory, practice, education and training*. Hrsg. von Matthew Lee, Ben-Zion Barta und Peter Juliff. Boston, MA: Springer US, 1995, S. 314–319. ISBN: 978-0-387-34848-3. DOI: 10.1007/978-0-387-34848-3_49. URL: https://doi.org/10.1007/978-0-387-34848-3_49.

[15] M. F. S. Oliveira u. a. "Software Quality Metrics and their Impact on Embedded Software". In: *2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*. 2008, S. 68–77. DOI: 10.1109/MOMPES.2008.11.

[16] M. O'Neal und W. Edwards. "Complexity Measures for Rule-Based Programs". In: *IEEE Transactions on Knowledge & Data Engineering* 6 (Okt. 1994), S. 669–680. ISSN: 1041-4347. DOI: 10.1109/69.317699. URL: doi.ieeecomputersociety.org/10.1109/69.317699.

[17] Paul Piwowarski. "A Nesting Level Complexity Measure". In: *SIGPLAN Not.* 17.9 (Sep. 1982), S. 44–50. ISSN: 0362-1340. DOI: 10.1145/947955.947960. URL: http://doi.acm.org/10.1145/947955.947960.