Legal Knowledge and Information Systems
G. Sileno et al. (Eds.)

doi:10.3233/FAIA230957

Insurance Portfolio Analysis as Containment Testing

Preston CARLSON [a] Michael GENESERETH [a]
[a] *Stanford University*

Abstract. Insurance Portfolio Analysis (IPA) is the process of comparing multiple, potentially overlapping insurance portfolios with an eye to detecting and characterizing redundancies and gaps in coverage. Unfortunately, insurees usually do not have the time or patience to compare policies from multiple insurance providers, and they often do not have the legal background needed to understand the complex legal wording of the contracts associated with those policies. Past work has shown that, by encoding policies as logic programs, it is possible to automatically determine compliance of specific claims with a policy's terms and conditions. In this paper, we show that it is also possible to automatically analyze multiple-policy portfolios for gaps and redundancies by assessing coverage over multiple hypothetical claims. We formalize the process of IPA and show how to use well-studied techniques for logic program containment testing to automate the process.

Keywords. insurance, computable contracts, logic programming

1. Introduction

We often think of insurance policies as being partitioned into distinct areas – home insurance, auto insurance, health insurance, travel insurance, and so forth; and we frequently buy different policies to provide coverage in these different areas. In reality, things are more complicated, with policies in different areas often providing overlapping coverage. For example, rental car damage may be covered by a policy purchased from a rental car company, a personal auto insurance policy, a credit card, a travel insurance policy, and even, in some cases, a home insurance policy. If we are unaware of these overlaps, we can end up paying more for insurance than we need; what's worse, there can be gaps between policies of which we are unaware.

Insurance Portfolio Analysis (IPA) is the process of comparing multiple, potentially overlapping insurance portfolios with an eye to detecting and characterizing redundancies and gaps in coverage [1]. For example, while renting a car, we may realize that we do not need to purchase collision insurance from Hertz because we are already covered if we use our Visa credit card. At the same time, we may realize that we need to purchase additional insurance for travel in Ireland, since our credit card insurance does not apply there.

The problem is that Insurance Portfolio Analysis is not easy. Insurees usually do not have the time or patience to compare policies from multiple insurance providers; and, even if they have the time, they often do not have the legal background needed to understand the complex legal wording of the lengthy, 100-page contracts associated with those policies.

The good news is that, with advances in information technology, it is now possible to implement computer systems to help automate this process. By representing insurance policies and insurees' coverage needs as logic programs, we can detect redundancies and gaps in coverage using the well-studied test of *logic program containment*.

This paper first presents a description of how insurance contracts are represented as logic programs, then discusses how such contracts are currently used to analyze specific insurance claims. After establishing this foundation, we introduce the primary analyses that comprise IPA, provide their formulations, and demonstrate how to analyze and compare *entire insurance portfolios* using techniques for containment testing. We conclude by discussing limitations of existing containment testing techniques, and future work.

2. Insurance Contracts as Logic Programs

Key to automating IPA is the encoding of insurance policies as computable contracts, and current computable contracts emphasize the automation of *claims analysis*. Such a computable contract seeks to determine the compliance of a *specific situation* with the terms and conditions of a *specific insurance policy*.

Most computable contracts are designed such that claims analysis can be performed via a standard evaluation of the program, and this is also true when contracts are encoded as logic programs. Accordingly, the input and output relations of these *logic program insurance contracts* have a consistent structure.

The input relations represent information that is relevant to determine coverage for a *specific claim*. (E.g. which policy it was filed under, where a specific hospitalization occurred, etc.) The set of output relations varies substantially between different computable insurance contracts, but it always includes a `covered` relation, which holds of a claim if-and-only-if the claim is covered by the policy. Additionally, the rules of the program represent policy-specific logic, and information about the world that is generally relevant to determining coverage, but which is not claim-specific. (E.g. geography, kinship relationships, general exclusions and conditions, etc.)

For this paper we follow two conventions. First, we frequently refer to a logic program insurance contract simply as a policy. Second, we treat the `covered` relation as unary, simply expressing the presence or absence of coverage for a claim. (This is in contrast to a binary relation, which could also express the quantity of payout for a given claim.) We do this for several reasons: (1) in practice, it is a prerequisite for determining the quantity of coverage since there is no payout if coverage is not present, (2) the analyses presented in later sections are more clearly stated when `covered` is unary, and (3) it is often more important to policyholders that their claim be covered *at all* by their policy than that the payout be a particular amount.

As a worked example of a policy, we encoded the Chubb hospital cash insurance policy from [2] as the following ruleset:

```
covered(C) :-
   policy(C,P) & active(P) & hospitalization(C,H) &
   validreason(H) & ~excluded(C)
active(P) :- signed(P) & paid_premium(P) & ~canceled(P)
validreason(H) :- reason(H,sickness)
validreason(H) :- reason(H,accidental_injury)
excluded(C) :- cause(C,skydiving)
excluded(C) :-
   hospitalization(C,H) & patient(H,X) & age(X,A) & A>=75
```

This policy determines coverage for a given claim by computing whether **(1)** the insurance policy is active, **(2)** the hospitalization referred to in the claim is for sickness or accidental injury, and **(3)** no exclusions apply to the claim. And, as expected, the input relations only represent information that is relevant to a specific claim and policy: `policy`, `hospitalization`, `signed`, `paid_premium`, `canceled`, `reason`, `cause`, `patient`, and `age`.

3. Claims Analysis

Once we have a logic program in the form described above, the formulation of claims analysis in the presence of complete information is straightforward. We provide a formulation of it here to better highlight the distinction between claims analysis and Insurance Portfolio Analysis.

Claims Analysis (Formulation): Consider an insurance claim `claim1`, and an input dataset D, where D is constructed such that it contains information about `claim1` that is relevant to the determination of whether coverage is present for `claim1`. Also consider P, a policy within which we would like to determine whether `claim1` is covered. Finally, P is executed on D and queried for the fact `covered(claim1)`.
Coverage is present for `claim1` under the policy P iff this query evaluates to true.

As an example, let's consider querying the policy from before, executed on the dataset below. It evaluates to true when queried for the fact `covered(claim1)`, indicating that the claim is covered. We emphasize that claims analysis as just discussed can be performed via a standard evaluation of the logic program.

```
policy(claim1, pol1)
hospitalization(claim1, hosp1)
cause(claim1, none)
signed(pol1)
paid_premium(pol1)
reason(hosp1, sickness)
patient(pol1, john_smith)
age(john_smith, 35)
```

4. Insurance Portfolio Analysis as Containment Testing

The goal of Insurance Portfolio Analysis (IPA) is to "detect and characterize redundancies and gaps in coverage". Doing so requires more than computing coverage of individual claims within individual policies — it requires reasoning about the *sets of claims* that are covered by *all of the policies* comprising a policyholder's insurance portfolio.

However, this cannot be done by simply enumerating and computing coverage for every possible claim, as there are infinitely many possible claims that could be filed under any given policy. Instead, we need to reason *symbolically* about the sets of claims that each policy can be evaluated on.

Fortunately, when computable insurance contracts take the form of logic programs, we can do exactly this via techniques for testing *logic program containment* — a *symbolic computation* that allows us to reason about the answers returned by logic programs when evaluated on arbitrary input databases.

Definition (Containment Testing): Consider two logic programs P_1 and P_2 over the same set of input relations. Let `query` be a distinguished output relation defined for both P_1 and P_2. Finally, consider an arbitrary input database D, and let $P(D)$ be the result of a program P executed on D. We say that P_1 is contained in P_2 (equivalently, P_2 contains P_1) iff $P_1(D) \subseteq P_2(D)$. That is, every answer returned by P_1 is also returned by P_2.

In the context of individual logic program insurance contracts, we take `covered` to be the distinguished output relation. Intuitively, this means that a policy P_2 contains another policy P_1 iff every claim covered by P_1 is also covered by P_2.

But we aren't limited to testing containment between individual policies! We can also test containment within an entire insurance portfolio by treating it as a union of logic programs. To do so, we represent an insurance portfolio as a set of policies $S = \{P_1, P_2, \cdots, P_n\}$ that all accept the same inputs, and all have a unary output relation `covered`. Then, the portfolio S covers a claim `claim1` iff any $P_i \in S$ covers `claim1`. Accordingly, the portfolio S contains a policy P iff every claim covered by P is also covered by S. (Checking this is not as simple as testing containment pairwise between P and each $P_i \in S$, because it may take multiple policies in S to cover all of the claims covered by P.)

Now, let's state the two analyses that constitute IPA in terms of sets of claims.

First, there is the determination of whether an insurance policy is redundant with the other policies in a portfolio. That is, we need to determine if *the set of situations for which a policy expresses coverage* is a subset of *the set of situations for which the rest of their insurance portfolio expresses coverage*.

Second, there is the determination of whether a policyholder's coverage needs are met by their insurance portfolio. That is, we need to determine whether *the set of situations for which the policyholder wants coverage* is a subset of *the set of situations for which their insurance portfolio expresses coverage*.

Once the insurance policies and the policyholder's coverage needs are encoded as logic programs, we can formulate these "Redundant Coverage" and "Coverage Needs" analyses as follows:

Redundant Coverage/Coverage Needs Analysis (Formulation):
Consider an insurance portfolio $S = \{P_1, P_2, \cdots, P_n\}$. Let P be a policy, and R be a program encoding coverage needs. Finally, let `covered` be the distinguished output relation in P, R, and in each $P_i \in S$.
We say that the coverage provided by P is *redundant with* that provided by S iff P is contained in S. Similarly, we say that the *coverage needs* expressed by R are met by S iff R is contained in S.

Note that since testing containment is necessary and sufficient for performing both analyses, any techniques developed for one analysis are directly applicable to the other!

As an example of Redundant Coverage Analysis, consider two auto insurance policies, Policy A and Policy B.

Policy A is encoded as the following rule, which expresses coverage for any claim in which the driver wasn't renting the vehicle mentioned in the claim.

```
covered(C) :- driver(C,D) & vehicle(C,V) & ~renting(D,V)
```

Policy B is encoded as the rules below, which express coverage for any claim in which the car wasn't moving (regardless of whether it was being rented) or in which the car was moving and was not being rented.

```
covered(C) :- vehicle(C,V) & ~moving(C,V)
covered(C) :-
    driver(C,D) & vehicle(C,V) & moving(C,V) & ~renting(D,V)
```

Via algorithms for containment testing, we can determine that Policy B *contains* Policy A, because any claim in which the driver wasn't renting the vehicle in the claim is also one in which the vehicle either wasn't moving, or was moving and was not being rented. Likewise, we can determine that Policy A *does not contain* Policy B, since Policy B covers claims that Policy A does not (namely, claims in which the vehicle was not moving but was being rented).

Existing techniques allow for testing containment between many expressive classes of logic programs. Testing containment between programs with no negation, views, or interpreted functions (Unions of Conjunctive Queries) is as simple as program evaluation [3]. Additionally, algorithms have been developed for when negations [4], arithmetic comparisons [5], general interpreted functions [6], or limited recursion [7] are permitted.

Extensions of these techniques allow us to test containment between programs with rules defined in terms of views (i.e. output relations), in many cases. When views aren't negated, we can flatten them into a union of single-rule queries and apply the algorithms above. When views *are* negated, we can often invert them into equivalent programs in terms of base relations. Unfortunately, we can't always invert negated views — for example, when rules are recursive. And, it is known that containment testing is not decidable for arbitrary logic programs [8]. Despite these limitations, all of the insurance contracts we have attempted to translate have been expressible as logic programs for which containment testing is decidable.

5. Conclusion and Future Work

This paper introduces the concept of Insurance Portfolio Analysis (IPA) and distinguishes IPA from traditional insurance claims analysis. It presents an approach to automating IPA using well-studied algorithms for testing logic program containment (by symbolic evaluation rather than explicit enumeration of claims). And it shows how this approach can be used to perform the two analyses that constitute IPA — Redundant Coverage Analysis and Coverage Needs Analysis.

Note that a key to our solution is the encoding of insurance policies as logic programs. This is preferable to encoding policies in traditional imperative programming languages (e.g. Java), since performing IPA requires determining program equivalence, and doing so with imperative programs is effectively impossible. This is likewise prefer-

able to directly processing insurance policies in natural language, as current NLP/LLM systems cannot answer even single-claim coverage questions "reliably and at-scale" [9].

Improving containment testing algorithms for more complex programs, especially those with interpreted predicates, would greatly improve the speed of Redundant Coverage and Coverage Needs analyses. And in order to accurately represent a user's coverage needs, UI/UX research must be done to allow users without programming experience to generate logic programs that express their needs. Furthermore, these analyses should be extended. Once it is determined that a policy is not redundant with, and that a policyholder's coverage needs are not met by, a portfolio, the overlaps (resp. gaps) in coverage should be characterized and explained to the policyholder. And, importantly, these analyses should be extended to cover programs that can compute the quantity of coverage in addition to its presence.

References

[1] Genesereth M. Insurance Portfolio Management [Internet]. 2022. Available from: https://law.stanford.edu/2022/07/30/insurance-portfolio-management/

[2] Goodenough O. Using "Toy Agreements" to Model Computable Contracts: Video Demonstration [Internet]. 2022. Available from: https://law.mit.edu/pub/usingtoyagreementstomodelcomputablecontractsvideodemonstr ation

[3] Ullman JD. Information integration using logical views. Theoretical Computer Science. 2000 May 28;239(2):189–210. doi:10.1016/s0304-3975(99)00219-4

[4] Mohamed KB, Leclére M, Mugnier M-L. Containment of Conjunctive Queries with Negation: Algorithms and Experiments. In: Bringas PG, Hameurlain A, Quirchmayr G, editors. Proceedings of Database and Expert Systems Applications, DEXA 2010; 2010 Aug 30-Sep 3; Bilbao, Spain. Berlin, Heidelberg: Springer; c2010. p. 330-45, doi:10.1007/978-3-642-15251-1_27

[5] Klug A. On conjunctive queries containing inequalities. Journal of the ACM. 1988;35(1):146–60. doi:10.1145/42267.42273

[6] Zhang X, Ozsoyoglu ZM. On efficient reasoning with implication constraints. In: Ceri S, Tanaka K, Tsur S, editors. Deductive and Object-Oriented Databases, DOOD 1993; 1993 Dec 6-8; Phoenix, AZ. Berlin, Heidelberg: Springer; c1993. p. 236–52, doi:10.1007/3-540-57530-8_15

[7] Calvanese D, De Giacomo G, Vardi MY. Decidable containment of recursive queries. Theoretical Computer Science. 2005;336(1):33–56. doi:10.1016/j.tcs.2004.10.031

[8] Shmueli O. Decidability and expressiveness aspects of logic queries. In: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '87; 1987 Mar 23-25; San Diego, CA. New York (NY): Association for Computing Machinery; c1987. p. 237–49, doi:10.1145/28659.28685

[9] Ancellin R, Carlson P, Doulcet P-L. Exploring Technologies for Automating Insurance Contracts Reasoning: A Beginner's Guide [Internet]. 2023. Available from: https://law.stanford.edu/2023/06/27/exploring-technologies-for-automating-insurance-contracts-reasoning-a-beginners-guide/