Transforming Domain Models to Efficient C# for the Danish Pension Industry

Holger Stadel Borum
hstb@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Morten Tychsen Clausen
mocl@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

ABSTRACT

Danish insurance and pension companies are required by financial regulations to report certain financial quantities to prove that they are solvent and managed responsibly. Parts of these quantities are computed the same way for all companies, whereas so-called management actions, describing, e.g., surplus sharing, vary between companies. Hence it is desirable to have a flexible calculation platform that allows actuaries to easily create company-specific models, which are also computationally efficient. In this paper, we present our work with implementing a code generator for a DSL called the Management Action Language (MAL) as a form of variability management. While one of the goals of MAL is to generate efficient code from an actuary's specification, it is non-trivial how to produce such code. We identify four reoccurring patterns in the models created by actuaries as subjects to optimisations. We describe our process for implementing a code-generator by a) identifying four specification patterns (inheritance, union types, type filtering, and numerical maps) that are pervasive in these calculations, and b) describing how to generate efficient C# from MAL for these patterns. We evaluate the code-generator by benchmarking it against handwritten production code and show an approximate 1.3× speedup in a production environment. This evaluation demonstrates that, with MAL, an individual pension company may reuse the general calculation platform and all of the optimisations built into MAL's code generator when modelling the company's business rules.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; **Domain specific languages**; • **General and reference** → **Performance**.

KEYWORDS

Domain specific languages, performance, code generation, vernacular software development

ACM Reference Format:
Holger Stadel Borum and Morten Tychsen Clausen. 2022. Transforming Domain Models to Efficient C# for the Danish Pension Industry. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3550356.3561580

1 INTRODUCTION

Current legislation requires Danish insurance and pension companies to report a number of quantities as a way of proving that they are solvent and managed responsibly [8]. The calculations of these quantities are complicated since they a) involve projecting financial products into the future, b) must model a specific company, c) are calculated in different economic scenarios, and d) involve complex mathematical formulas. While parts of these calculations are identical for all companies (such as the projection of assets and reserves), other parts are company-specific since they model company-specific rules called management actions (such as surplus sharing). This duality means that a projection platform consists of a generic company-independent part and a specialised company-specific part, as shown in Figure 1.

In the current practice, actuaries submit management action specifications to a projection platform written in a general-purpose language such as C#, as seen in Figure 1. This practice is challenged by actuaries with limited programming experience (or *vernacular software developers* [20]) being required to specify *efficient* management actions due to projections being computationally expensive.

Our proposal is to modify the current practice with a domain-specific language (DSL) and a code generator that generates efficient management actions from more declarative specifications. Our DSL called the Management Action Language (MAL, Section 2), was designed for the purpose of allowing actuaries as vernacular software developers to model management actions and have them automatically transformed, optimised, and executed efficiently on a projection platform. In this paper, we present our work with implementing a code generator for MAL to perform this transformation (Section 3). We identify four specification patterns that we find are pervasive in management action specifications and discuss how we generate efficient code for these in a target language with a more restrictive type system (Section 3.1-3.3).

In doing so, we demonstrate a case where a DSL can be used to generate code that is 1.3× more efficient compared to handwritten production code in the target language (Section 4).

In short, the contributions of this paper are:

- An identification of specification patterns pervasive in management action specifications.
- Implementation strategy for generating code from these specifications with C# as target language.
- Positive results that show a speedup when benchmarking generated code against comparable handwritten code.

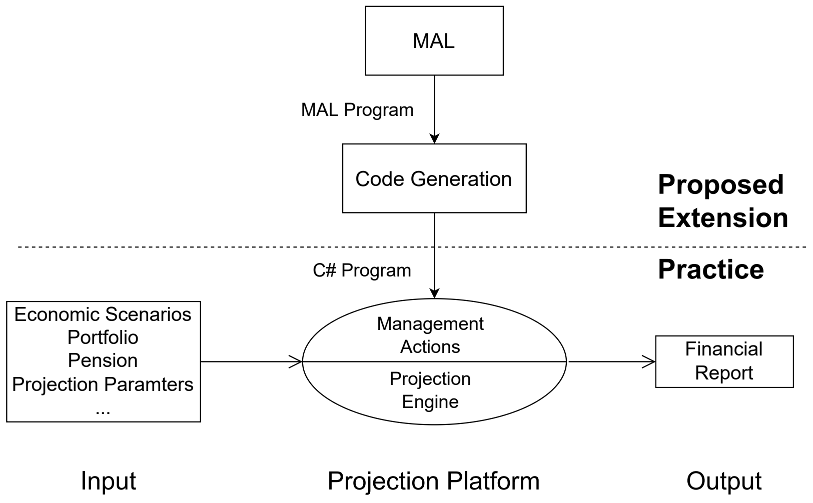


Figure 1: A diagram with the current projection process where the projection platform is separated into a general *projection engine* and company-specific *management actions*. In the proposed extension, the management actions are generated from a specification written in the Management Action Language.

- Negative results that do not show a speedup for loop vectorisation.

2 THE DANISH PENSION INDUSTRY AND DSLS

The Danish pension and life insurance industry manages a large number of assets [12], and it is, therefore, important for the Danish economy that the industry is healthy. Here we give an ultra-compact conceptual model of the domain that is coherent with the DSL concepts described in the paper, primarily based on [17]. A *pension* (or *life insurance*) is a saving to be paid to the *policyholder* (or a beneficiary) upon the policyholder reaching a specific age (or dying). A *pension company* is a company that offers pension and life insurance *products* to customers. A *policy* is an instance of a product bought by a specific customer. The policies held by a company are, collectively, the *portfolio* of the company. The *reserve* of a policy is its present value which is the total of all its future payments. A company is *solvent* when it can meet all expected future insurance claims. A company determines whether it is solvent by computing its *balance* consisting of its *assets*, e.g., bonds, and *liabilities*, where the reserve of policies is a substantial part of the liabilities. It is common to put a policy into different types of *groups* according to its *interest* rate, biometric *risk*, and administrative *expense* profile.

The projection of the balance is one way for a pension company to argue that it is responsibly managed and will remain solvent. A pension company is required by Danish regulation to make such an argument to the Danish Financial Supervisory Authority (FSA). There are two major challenges in performing such a projection which we call the *performance* challenge and *variability* challenge. The performance challenge is that such projections are computationally expensive. A projection should take many different economic scenarios (in the order of 10 up to 1000) into account to simulate how the portfolio behaves with varying yields and losses on investments. One way of doing so is to parametrise a projection on an economic scenario and then perform a Monte Carlo simulation over different economic scenarios. This simulation approach is the one used by the projection platform that MAL was designed for. The variability challenge is that a projection must model certain business rules of the company that specify how investment yields and losses are generally handled. These business rules are called management actions. Management actions are not used only as input to a balance projection but must also be shown and approved by the board of directors of a pension company and be reported with "recipe-like" precision to the FSA.

The Management Action Language was designed to allow actuaries to model management action in balance projections. A design goal was to emphasise the mathematics of these models while minimising boilerplate code. MAL consists of three parts: On top, a module system lets actuaries group management actions into meaningful computational units. Each module consists of two parts: First, data declarations and data contracts define what data a module requires and provides (see Figure 3 for a small example of data declarations). Second, imperative management actions and pure functional expressions define how a module computes data. The design choice behind having expressions without side effects (i.e. no increment expression `i++`) is one of always being able to safely parallelize the computation of multiple expressions. Side effects are still present in the language, however only at the management action level.

Figure 2 shows a fragment of a MAL module that uses computational patterns that are pervasive when modelling management actions.

3 CODE GENERATION

In this section, we give a high-level description of our approach to translating MAL into C#. The translation is subject to the following somewhat conflicting design considerations.

```
update eGroup in Groups:Expense {
  let reserveInclIRTA = eGroup.Reserve + eGroup.PAL.InvestmentReturnTaxAsset
  let periodTechnicalExpenses =
    sum( map p1 in eGroup.Policies:OneStatePolicy {
           Util.SumExpenses(p1.Result.PeriodTechnicalExpenses)
         }
       )
  eGroup.ExpenseDividend =
      if periodTechnicalExpenses == 0
      then 0
      else bound(0.02 * reserveInclIRTA / (periodTechnicalExpenses / Projection.PeriodLength), 0, 1)
}
```

Figure 2: An abridged but realistic MAL fragment demonstrates commonly used patterns, namely: a) inheritance based type filtering of `Expense` and `OneStatePolicy` and b) a often used map-sum pattern.

(1) C# was chosen as the target language for the code generator due to a wish to potentially generate a self-contained C# version of a MAL program.
(2) The produced C# should be compatible with several existing interfaces and flexible enough to handle some evolution of these.
(3) Performance of the generated code was important due to the computational cost of calculations.

During the design of MAL, we identified four specification patterns (inheritance, union types, type filtering, and numerical maps) that were pervasive in expressing management actions. As an example, there may exist three different kinds of Groups (see Section 2), say Risk, Interest, and Expense (object-oriented inheritance). Both Risk and Expense groups may have a shared quantity we want to compute (union type). We may compute the value by choosing all Risk or Expense groups (type filtering) and then, for each group, compute the value specified by summing over a map of the group's policies (numerical map). We describe how we generate C# code for each of the above-specified patterns. To accommodate type filtering, we generate what we call a *TypeSpan*, which is a collection of objects with constant time type filtering. In Section 4, we argue the generated code is comparatively efficient and correct.

Figure 3 shows an example of how we translate data declarations in MAL to classes and interfaces in C# using the subsequently presented translation schemes. However, to conserve space, we will primarily work on an abstract representation of a MAL program without presenting the full abstract or concrete syntax. To do so, given a MAL program, we define $\mathtt{L}$ to be a set of labels of data fields used, $\mathtt{T}$ to be the set of the program's types, $\mathtt{T}[\![\_]\!]$ to be a type translation from MAL to C#, and $<:$ to be a subtype relation on $\mathtt{T}$ with $\mathtt{lub}^{<:}(\_)$ as a function providing the least upper bound on a set of types. We use $\mathrm{dom}(\_)$ to denote the domain of a function, $\rightharpoonup$ to denote a partial map, and $\mathbb{P}(\_)$ to denote the powerset.

3.1 Data Declarations and Inheritance

MAL implements a single inheritance system that affords users the possibility of describing different versions of the same actuarial concept. That is, if a user wants to create a Risk group and an Expense group, they can do so by letting both inherit from Group.

Definition 1. The data declarations of a MAL program can be represented by the tuple $(\mathtt{D}, \mathtt{I}, \mathtt{F})$ where:

- $\mathtt{D}$ is a set of data declaration names.
- $\mathtt{I}$ is a binary relation on $(d_1, d_2) \in \mathtt{D}$ denoting that d_1 inherits from d_2. $\mathtt{I}^*$ denotes the transitive closure on $\mathtt{I}$.
- $\mathtt{F} : \mathtt{D} \longrightarrow \mathtt{L} \rightharpoonup \mathtt{T}$ is a map from a data declaration name to its fields to their type.

We only consider well-formed data declaration where (1) a data declaration at most inherits from a single other declaration and (2) an inheritance only specialises fields from its supertype while respecting the subtype relation, or formally:

- $\forall (d_1, d_2) \in \mathtt{I} \,.\, \forall d_3 \in \mathtt{D} \,.\, d_2 \neq d_3 \Rightarrow (d_1, d_3) \notin \mathtt{I}$
- $\forall (d_1, d_2) \in \mathtt{I} \,.\, \forall lb \in \mathrm{dom}(\mathtt{F}(d_2)) \Rightarrow \mathtt{F}(d_1)(lb) <: \mathtt{F}(d_2)(lb)$

Figure 4 shows a scheme $\mathtt{D}[\![\_]\!]$ for translating data declarations into C#. The primary thing to notice about the scheme is that when a data declaration is extended with a field that is a subtype of the original field, then this field is shadowed in the generated subclass. This choice means that in MAL, there are severe restrictions to field assignments on a superclass to ensure that the generated class hierarchy remains consistent throughout an execution.

3.2 Union Types and TypeSpans

To provide users with the possibility of working flexibly with collections of entities, MAL implements data declaration filtering on collections with the possibility of doing so with a union type. Furthermore, by using our notion of TypeSpans this filtering may happen in constant time.

Definition 2. The union types used in a MAL program are denoted by $\mathtt{U} \subseteq \mathbb{P}(\mathtt{D})$. We assume we have an injective function $id_{\mathtt{U}} : \mathbb{P}(\mathtt{D}) \longrightarrow \mathtt{String}$ that provides a union type with a unique name. The function $\mathtt{F}_{\mathtt{U}} : \mathbb{P}(\mathtt{D}) \longrightarrow \mathtt{L} \rightharpoonup \mathtt{T}$ provides types to the fields of $u : \mathtt{U}$ with $\mathtt{F}_{\mathtt{U}}(u)(l) = \mathrm{lub}^{<:}(\{\mathtt{F}(d)(l) | d \in u\})$. The function $\mathtt{F}_{\mathtt{U}}(u)$ is only defined on the domain $\cap_{d \in u} \mathrm{dom}(\mathtt{F}(d))$.

A union type $u : \mathtt{U}$ is well-formed when it (1) is non-empty and (2) all names within the union inherit transitively from a given data declaration, or formally:

- $u \neq \emptyset$
- $\exists d' \in \mathtt{D} \,.\, \forall d \in u \,.\, (d, d') \in \mathtt{I}^*$

For each union type of a MAL program, we generate an interface corresponding to the type (Figure 5) and a TypeSpan that serves as a collection for the union type with constant time filter operations (Figure 5). While we potentially need to make this generation for exponentially many combinations of a data declaration's subtypes,

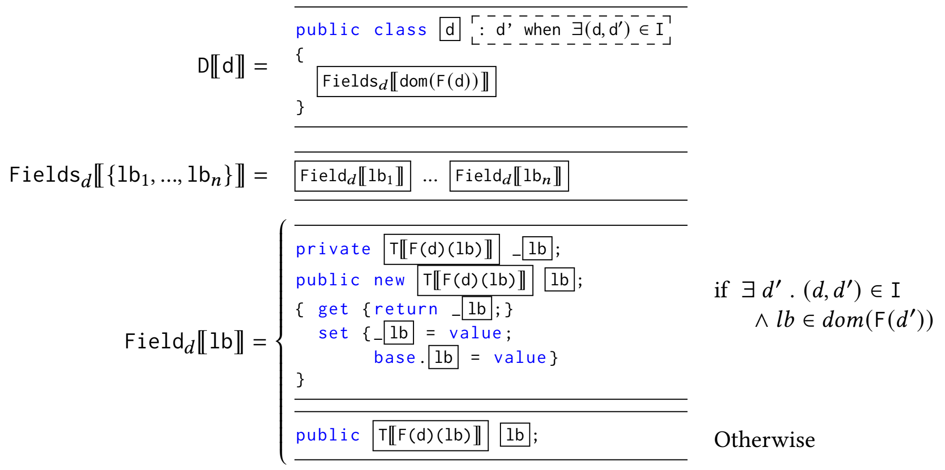
```
data Group {
  Child : Group
}
data Risk extends Group {
  Child : Risk
}
data Expense extends Group {
}
```

```
public class Risk : Group,Expense_Risk {
  private Risk _Child;
  public new Risk Child
  { get {return _Child;}
    set {_Child = value;
         base.Child = value;}
  }
}
```

```
public interface Expense_Risk {
  Group Child { get; set; }
}
```

```
public class TypeSpan_Expense_Risk : IEnumerable<Expense_Risk> {
  private Expense[] Expense;
  private Risk[] Risk;
  public TypeSpan_Expense Filter_Expense() {
    return new TypeSpan_Expense(Expense);
  }
  /* Remaining Filters and enumeration omitted. */
}
```

Figure 3: Examples of code generation. Top left: An artifical MAL-snippet with a data declaration for `Group` and the two subtypes `Risk` and `Expense`. Top Right: Generated C# class for `Risk`. Middle: Generated C# interface for the `Expense` and `Risk` union type. Bottom: The generated C# TypeSpan that is a collection type for `Expense` and `Risk` with constant time filtering.



```
D⟦d⟧ =            public class [d] ⌜: d' when ∃(d, d') ∈ I⌝
                  {
                    [Fields_d⟦dom(F(d))⟧]
                  }

Fields_d⟦{lb_1, ..., lb_n}⟧ =   [Field_d⟦lb_1⟧] ... [Field_d⟦lb_n⟧]

Field_d⟦lb⟧ = {   private [T⟦F(d)(lb)⟧] _[lb];
                  public new [T⟦F(d)(lb)⟧] [lb];
                  { get {return _[lb];}
                    set {_[lb] = value;
                         base.[lb] = value}
                  }                                  if ∃ d' . (d, d') ∈ I
                                                        ∧ lb ∈ dom(F(d'))

                  public [T⟦F(d)(lb)⟧] [lb];         Otherwise
```

Figure 4: Translation scheme, $\mathtt{D}⟦\_⟧$, for generating classes from data declarations. The scheme uses $\mathtt{Fields}_d⟦\_⟧$ to generate a field for all labels (lb) of a data declaration using $\mathtt{Field}_d⟦\_⟧$. Notice how $\mathtt{Field}⟦\_⟧$ shadows a field when a data declaration specialises the type of the field. Legend: A solid frame $\boxed{}$ delimits a quasi-quotation area where we may make use of our functions, variables, and some pseudocode. A dashed frame ⌜ ⌟ delimits code parts that are only included when relevant.

we can limit this number to the union types that are actually used in a concrete MAL program. This means that while the size of the generation is theoretically upper bounded by an exponential function, it is linear in the length of a program. The generation of TypeSpan provides users with a constant filter operation on a collection of data declarations, with zero cost for type casts or type checking.

3.3 Vectorisation

The expression language of MAL contains a map operation since it is frequently used in management specifications. When generating code for a numerical map operation, it seemed like an obvious optimisation to generate code using advanced vector extensions (AVX). We call this form of generation *vectorisation.*

Our approach to vectorisation was to vectorise all map operations that contained an arithmetic operation, which is always possible since MAL's expression language is free of side effects. Since most values are fields on an object in MAL, we had to transform relevant values into AVX vectors when needed. We used the following high-level process of vectorisation:

(1) Identify that a map operation uses an arithmetic operation.

```
U⟦u⟧ =                      public interface idU(u) {
                              UFieldsu⟦dom(F(u))⟧
                            }

UFieldsu⟦{lb1, ..., lbn}⟧ = T⟦Fu(u)(lb1)⟧ lb1 { get; set; }
                            ... public T⟦Fu(u)(lbn)⟧ lbn { get; set; }
```

```
TS⟦{d1, ..., dn}⟧ =         public class TypeSpan_idu({d1, ..., dn})
                                  : IEnumerable<idu({d1, ..., dn})> {
                              private T⟦d1⟧[] d1;
                              ... private T⟦dn⟧[] dn;

                              For all u′ ∈ ℙ({d1, ..., dn}) generate Filter⟦u′⟧
                              // constructors, enumeration, etc. omitted
                            }

Filter⟦{d1, ..., dn}⟧ =     public TypeSpan_idU({d1, ..., dn}) Filter_idU({d1, ..., dn})() {
                              return new TypeSpan_idU({d1, ..., dn})(d1, ..., dn);
                            }
```

Figure 5: On top: Translation scheme, U⟦_⟧ , for generating interfaces for union types. The scheme uses UFS$_u$⟦_⟧ to generate the fields on the union type. On bottom: Translation scheme for generating the TypeSpan for a union type. Note the constant time filtering operations created by Filter⟦_⟧.

(2) Traverse the collection being mapped over in slices of the hardware-specific size of the AVX registers.
(3) Transform relevant data to AVX vectors.
(4) Use AVX instructions whenever possible.

However, in an actual implementation, several other details show up to ensure efficient code, such as an identification of loop-constant expressions, removal of intermediate arrays, compatibility with deforestation [25], and improving heuristics for choosing when to vectorise an expression.

The idea behind the optimisation is that an AVX instruction is used on $2 * \texttt{vl}$ elements rather than just 2 at a time, where vl is the hardware-specific register size. The total number of operations performed should thus become $\sim\frac{1}{\texttt{vl}}$ of the number of operations in a non-vectorised translation.

4 EVALUATION

We evaluate MAL's implementation from a technical perspective since a user-oriented evaluation has previously been conducted [7]. We evaluate by comparing the performance of generated code against an equivalent handwritten specification written in C#. This specification is realistic in the sense that significant effort has been put into its over 14,000 lines of code while being developed with the entire projection platform by a combination of actuaries and software developers. It is also realistic since it serves as a template implementation for several pension companies who may either modify the template or write their own implementation from scratch.

4.1 Performance Experiments

We are interested in investigating how the performance of code generated from MAL compares to comparative handwritten C# code. For performance measures, two different benchmark setups are used. The first local setup almost executes only management code on a dedicated benchmark machine to investigate how the generated code performs. The second production setup executes a full projection in a realistic execution on a cloud service to investigate the total impact of using MAL in the real system.

4.1.1 Only management code. We are interested in a) how does MAL perform on a portfolio of realistic size? And b) how does the MAL implementation scale on increasing portfolio sizes? The benchmark was performed on a machine with an Intel Xeon E5-2680 v3 with 48 logical 2.5 GHz cores and 32 GB of memory, running 64-bit Windows 10, version 20H2, and .NET 4.8.4084. The result of the benchmarks is seen in Figure 6.

Observation 1. *The execution time of all solutions scales linearly in the number of policies.*

We explain this observation from the fact that all solutions perform their majority of work linearly in the number of exercises.

Observation 2. *For all number of policies, the solution generated from a MAL program is faster than the C# solution with a speedup factor of approximately 1.3×.*

We explain this speedup from the second observation from the fact that: a) MAL's translation from data declarations to classes provides a pretty efficient memory layout compared to the C# solution, which often uses dictionaries and indirect lookups for

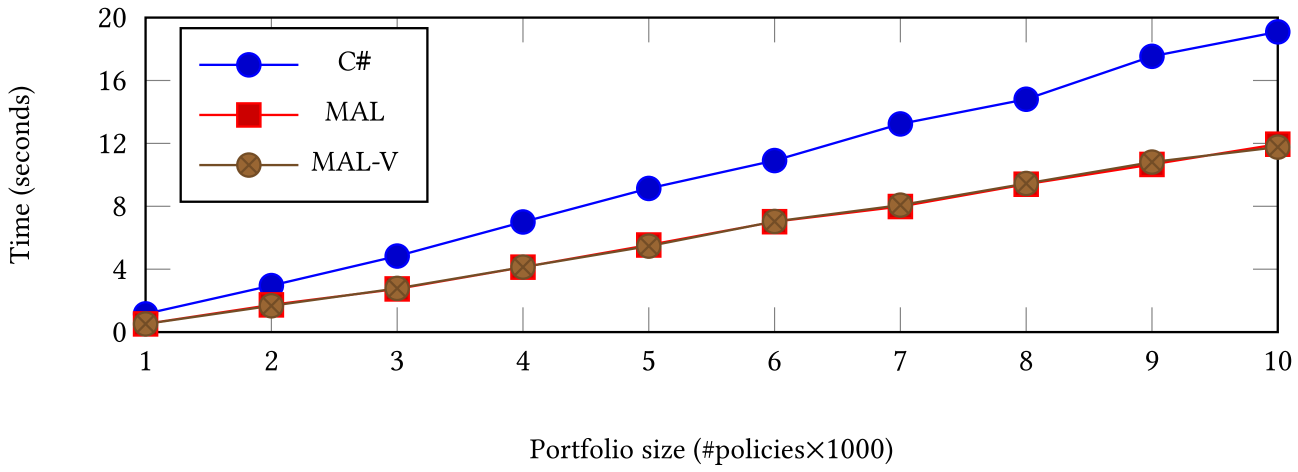


Figure 6: Running time of benchmarking management code performed on randomly generated portfolios. The benchmarks were generated using BenchmarkDotNet [2] with `WarmupCount=8, IterationCount=40, InvocationCount=1`. One standard deviation is plotted as the error but it is not visible since it is generally $< 3\%$.

similar purposes. b) MAL solution generates less garbage due to the use of classes and deforestation. c) MAL's easily reusable constant time type filtering without type casts is faster than checking every individual of a collection. Although this functionality is possible to write in C# it is difficult to do in any reusable manner without some form of code generation.

Observation 3. We cannot find any difference in the execution time of code generated with and without vectorisation.

We explain this negative result from the overhead of taking numerical values from an object and placing them into an array. We hypothesise that we need a more vectorisation-friendly memory layout to see a speedup.

4.1.2 Full projection. We are interested in how using MAL affects the part of the execution that does not involve executing management code directly. However unlikely, it is possible that MAL's memory footprint, some delayed execution, or slower serialisation means that the full projection using MAL could be slower, even if the management code executes faster. For full projections, it is worth noting that even though the majority of execution time takes place outside of management code, then the time spent in management has grown significantly (from around 10% to around 20%). It seems like the trend is likely to continue for two reasons. First, parts of the non-management code may be optimised for all projections (e.g., by solving some partial-differential equitations analytically instead of numerically). Second, new requirements to the projection platform have a tendency to be implemented as management code that is available for users to modify. Table 1 shows the speedup from using MAL in a production setting. From this benchmark, we conclude that MAL provides around 1.3× speedup compared to handwritten management actions and that MAL does not negatively affect the remainder of the projection. The setup and data used for this benchmark are quite different from that used in Figure 6, which means that their real times are not comparable.

4.2 Implementation Correctness

We argue that MAL's code generator is correct by passing a test suite that consists of the following four kinds of tests:

(1) MAL passes 9 relevant regression tests designed for the C#-solution. The only test that does not pass requires some extra functionality that is currently not present in MAL.
(2) MAL produces the same results as the code for C# randomly generated portfolios.
(3) Using FSCheck [1] to create a generator for random valid MAL programs, we have performed a series of property-based tests. The properties we tested for are:
    - Printing and then parsing a MAL program results in a syntactically equivalent program.
    - A typed MAL program retains its types after printing and parsing it.
    - Transforming an error-free MAL program produces an error-free C# program.
(4) A small suite of unit tests ensures the correctness of different smaller components.

5 RELATED WORK

Danish pension companies must perform balance projections based on management actions, and therefore these companies all need an executable formalisation of their management actions, whatever platform they use. While it is common for actuaries at Danish pension companies to publish their actuarial models for academic inspection and discussion, it is not common for Danish pension companies to publish information on their software solutions. However, from participating in project advisory board meetings and from an industry conference, we find that Danish pension companies specify their management actions in general-purpose languages

Table 1: Benchmarks for full projections in a production environment on *Standard_F2s_v2* machines. The table shows two different projection setups with speedups for only management code and for the full projection.

	10k policies, 1 economic scenario			1k policies, 1k economic scenarios		
Task	C#	MAL	Speedup	C#	MAL	Speedup
Management Init	1.0 s	3.8 s	0.2×	238.8 s	469.9 s	0.51×
Management Update	26.4 s	17.3 s	1.77×	3198.0 s	1833.7 s	1.74×
Management Finalize	1.5 s	1.7 s	1.15×	123.3 s	131.5 s	0.94×
Management Total	**28.9 s**	**22.8 s**	**1.27×**	**3,560 s**	**2,435 s**	**1.46×**
Full Projection Total	**120.3 s**	**109.8 s**	**1.09×**	**15,943 s**	**14,486 s**	**1.10×**

(C# and Visual Basic) but are interested in other approaches. To our knowledge, our project is the only DSL that lets actuaries express management actions, and this article is the first on the efficient execution of management actions.

Domain-specific languages have been used several times in financial domains to model financial products or entities, e.g., [18, 22–24]. MAL is a spiritual successor to the Actuarial Modeling Language (AML) [9] that is used to model pension products. MAL is somewhat different from these DSLs since it does not seek to precisely model financial products but rather the management or manipulation of financial products. In this aspect, MAL is similar to the proprietary actuarial DSL T# [3], which is part of the company RPC's financial modelling platform called Tyche. While there is little public documentation of T#, the language approach is different from ours since it is a scripting language that seemingly calls into an external library.

Other DSLs from vastly different domains with different target languages. For example, OptiML is designed to let machine learning researchers specify machine learning algorithms that are translated to efficient parallel code [21], SARVAVID is designed to ease genomic analysis by providing kernels for doing so and generating efficient C++ [15], and CFDLang is designed for specifying performance-critical operations in computational fluid dynamics can be translated into efficient C code [13, 19].

Many aspects used in MAL's code-generation design are covered generally and in-depth by others. For object-oriented inheritance, see [4], for union types see [11], and for vectorisation, see [14]. We considered looking into optimisations for loop parallelism [16] and data flattening [5] similar to the parallel language of Futhark [10], but found a conflict in the design of MAL compared to Futhark, which hindered such an approach. Futhark is a functional language where programs are built from parallel primitives operating on arrays of data. MAL is designed with actuaries as target users and leans toward object-oriented programming to be more reminiscent of C#, as actuaries in the Danish industry are most likely to have some limited experience with C# or Java. Users of MAL are thus not limited to programming using parallel building blocks, and any guarantee of performance increase from automatic parallelisation is lost.

TypeScript is a widely-used general-purpose language that implements the concept of union types [6], but with JavaScript as its target language, union types can seemingly just be erased when translating to JavaScript. Our work is different since we have to embed union types in the type system of C#.

6 CONCLUSION

In this paper, we have presented our work with generating efficient management actions for the pension industry from MAL. We identified four specification patterns that we found pervasive in the specification of management actions, namely: inheritance, union types, type filtering, and numerical maps. We described our efforts with generating efficient C# code for these patterns by showing translation schemes. We have shown that the code generated from our translation schemes leads to an approximate 1.3× speedup and conjecture that a significant part of this comes from our work with interfaces for union types and a collection with a constant time type filtering. We also report a negative result: our effort to generate code for numerical maps using AVX instruction resulted in no speedup.

Acknowledgements. We thank Innovation Fund Denmark (7076-00029B) for funding this work. We also thank Peter Sestoft for supervising this work.

REFERENCES

[1] FsCheck: Random Testing for .NET. https://fscheck.github.io/FsCheck/. Accessed July 2022.
[2] BenchmarkDotNet, 2021. https://benchmarkdotnet.org/. Accessed Dec 2021.
[3] RPC Tyche - Tyche, 2021. https://www.rpc-tyche.com/products/tyche.html. Accessed Dec 2021.
[4] Abadi, M., and Cardelli, L. *A Theory of Objects.* Monographs in Computer Science. Springer New York, New York, NY, 1996.
[5] Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., Rosen, S., and Shaw, A. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (Shenzhen, China, Feb. 2013), PPoPP '13, Association for Computing Machinery, pp. 81–92.
[6] Bierman, G., Abadi, M., and Torgersen, M. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming* (Berlin, Heidelberg, 2014), R. Jones, Ed., Lecture Notes in Computer Science, Springer, pp. 257–281.
[7] Borum, H. S., Niss, H., and Sestoft, P. On Designing Applied DSLs for Non-programming Experts in Evolving Domains. In *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*

(Virtual Event, Fukuoka, Oct. 2021), MODELS '21, Association for Computing Machinery.

[8] Bruhn, K., and Lollike, A. S. Retrospective reserves and bonus. *Scandinavian Actuarial Journal* (Aug. 2020), 1–19.

[9] Christiansen, D., Grue, K., Niss, H., Sestoft, P., and Sigtryggsson, K. S. An Actuarial Programming Language for Life Insurance and Pensions. In *Proceedings of 30th International Congress of Actuaries* (2013).

[10] Henriksen, T., Serup, N. G. W., Elsman, M., Henglein, F., and Oancea, C. E. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 556–571.

[11] Igarashi, A., and Nagira, H. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing* (Dijon, France, Apr. 2006), SAC '06, Association for Computing Machinery, pp. 1435–1441.

[12] Jensen, B. M., Raffnsøe, M. D., and She, J. Forsikrings- og pensionssektoren i ny kvartalsvis statistik, 2019. (In English: The Insurance Sector and Pension Sector in New Quarterly Annualy Statistic).

[13] Karl Friebel, F. A., Soldavini, S., Hempel, G., Pilato, C., and Castrillon, J. From Domain-Specific Languages to Memory-Optimized Accelerators for Fluid Dynamics. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept. 2021), pp. 759–766. ISSN: 2168-9253.

[14] Kennedy, K., and Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[15] Mahadik, K., Wright, C., Zhang, J., Kulkarni, M., Bagchi, S., and Chaterji, S. SARVAVID: A Domain Specific Language for Developing Scalable Computational Genomics Applications. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey, June 2016), ICS '16, Association for Computing Machinery, pp. 1–12.

[16] Oancea, C. E., and Rauchwerger, L. Logical inference techniques for loop parallelization. *ACM SIGPLAN Notices 47*, 6 (June 2012), 509–520.

[17] Perdersen, L. R. *Grundlæggende firmapension*, 5 ed. Forsikringsakademiets Forlag, 2014. (In English: Foundation in Company Pensions).

[18] Peyton Jones, S., Eber, J.-M., and Seward, J. Composing contracts: an adventure in financial engineering (functional pearl). *ACM SIGPLAN Notices 35*, 9 (Sept. 2000), 280–292.

[19] Rink, N. A., Huismann, I., Susungi, A., Castrillon, J., Stiller, J., Fröhlich, J., and Tadonki, C. CFDlang: High-level code generation for high-order methods in fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018* (Vienna, Austria, Feb. 2018), RWDSL2018, Association for Computing Machinery, pp. 1–10.

[20] Shaw, M. Myths and mythconceptions: what does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages 4*, HOPL (Apr. 2022), 234:1–234:44.

[21] Sujeeth, A. K., Lee, H., Brown, K. J., Chafi, H., Wu, M., Atreya, A. R., Olukotun, K., Rompf, T., and Odersky, M. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (Bellevue, Washington, USA, June 2011), ICML'11, Omnipress, pp. 609–616.

[22] van Deursen, A. Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study, 1997.

[23] Voelter, M. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, Lexington, KY, Jan. 2013.

[24] Voelter, M., Koščejev, S., Riedel, M., Deitsch, A., and Hinkelmann, A. A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV. In *Domain-Specific Languages in Practice*, A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, Eds. Springer International Publishing, Cham, 2021, pp. 93–130.

[25] Wadler, P. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science 73*, 2 (Jan. 1988), 231–248.