Actulus Modeling Language

– An actuarial programming language for life insurance and pensions.

This paper illustrates how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation. The notation is human-readable and machine-processable, and specialised to the actuarial domain, achieving great expressive power combined with ease of use and safety.

edlund
– inside insurance

An Actuarial Programming Language for Life Insurance and Pensions

David R. Christiansen[1], Klaus Grue[2], Henning Niss[2], Peter Sestoft[1], and Kristján S. Sigtryggsson[2]

[1]IT University of Copenhagen, Denmark*
[2]Edlund A/S, Denmark

October 3, 2013

Contents

*Work supported by the Danish Advanced Technology Foundation (Højteknologifonden) (017-2010-3)

Abstract

We show how the design of pension and life insurance products, and their administration, reserve calculations, and audit, can be based on a common formal notation.

This notation is human-readable and machine-processable, and specialized to the actuarial domain, achieving great expressive power combined with ease of use and safety. In essence, this is a specialized actuarial programming language, a so-called domain-specific language. The language comprises (a) product definitions based on standard actuarial models, including arbitrary continuous-time Markov and semi-Markov models, with cyclic transitions permitted; (b) calculation descriptions for reserves and other quantities of interest, based on differential equations; and (c) administration rules.

1 Introduction

Our vision is to enable a formalized description of life insurance and pension products that supports automated administration and reporting, yet still is readable and manageable by humans. This should ensure consistency between all company operations: distributing incoming payments across coverages, generating annual statements for customers, paying benefits, calculating expected net present values (that is, reserves), producing reports for accounting and tax purposes, and so forth. Quantitative aspects such as reserves are calculated by a calculation kernel, incorporating high-performance numerical differential equation solvers.

The approach taken is to design and implement a domain-specific language, called the Actulus Modeling Language (AML) to describe life-based pension or life insurance products, and computations on them.

Because of the specialized nature of the language and its direct basis in actuarial theory, we can provide automated tools for early detection of certain kinds of errors and inconsistencies in the design of an insurance product. Moreover, even though the notation reflects a high-level actuarial view of products and reserve calculations, the calculations can be optimized automatically. The short distance from actuarial thought and notation to efficient computation enables much faster and less expensive development of new pension products.

1.1 Contributions

We present a domain-specific formal language in which actuaries can describe life insurance and pension products. This language relies on a calculation kernel, a general software framework that can express and perform a range of computations on such products, for instance to compute reserves, stress scenarios and cash flows.

We split what is specific to a given product (this is described in the domain-specific language) from what is common to all products (this is implemented by the calculation kernel). Via this split we expect to avoid the intricate and opaque program code typically tailor-made for each insurance product in IT systems for life insurance and pension administration.

Moreover, we achieve coherence across the entire life insurance company. Actuaries can use the domain-specific language (and the calculation kernel) for rapid experiments with new product designs, and the exact same product description can then be used in subsequent administration, reporting, solvency computations, and so on.

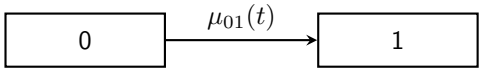
The separation into a specific product description and a general calculation kernel is expected to bring long-term benefits to the development and maintenance of the IT infrastructure of life insurance companies. For instance, when the underlying technological platform (operating system, compute clusters, cloud computing, programming languages, database systems) evolves, this affects the implementation of the calculation kernel only; the product descriptions are not affected at all. This simplifies software evolution considerably, compared with the current situation in which products are implemented as scattered company-specific adaptations of a common software code basis. It will no longer be necessary to perform error-prone changes to multiple copies of the common code base when the technological platform changes.

1.2 Actuarial concepts and notation

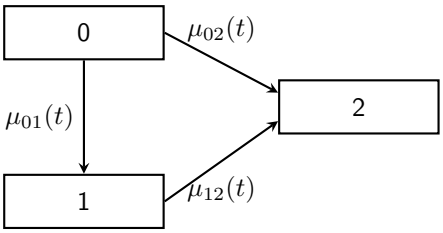
The AML system uses continuous-time Markov models for life insurance and pension products; these are more general than discrete-time state models. In this section, we informally describe the models and terminology that underlie AML.

A continuous-time Markov model consists of a finite number of *states*, typically denoted by the numbers $0, 1, 2, \ldots$ and *transition intensities* between these states. The transition intensity $\mu_{ij}(t)$ from state i to state j, when integrated over a time interval, gives the probability that a transition from state i to state j will occur in the interval. These models exhibit the *Markov property*, which is to say that future transitions depend on the past only through the current state. In some cases, we also allow definitions based on *semi-Markov models*, in which the transition intensities can additionally depend on the duration of sojourn in the source state.

A simple two-state Markov model, as seen in Figure 1a, can be used to represent the mortality of a single person. State 0 represents that the insured



(a) Two-state mortality model



(b) Three-state disability model

Figure 1: State models

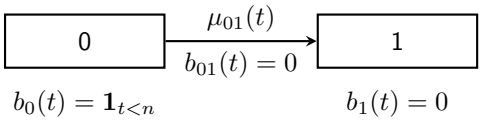
is alive, while state 1 represents that he or she is dead. The mortality intensity $\mu_{01}(t)$ represents the rate of mortality for the insured, which will be determined from information such as the age, sex, and occupation of the insured.

A slightly more complicated model must be used for products offering disability insurance. These products can be modeled with three states, representing active labor market participation, disability that precludes employment, and death, respectively represented as 0, 1 and 2 (Figure 1b). There are transitions from active participation to disability and to death, and from disability to death. Additionally, some products may allow for *reactivation*, where a previously disabled customer begins active labor again. Many current actuarial systems are restricted to acyclic models. Because the AML system uses a numerical differential equation solver, it can be used with cyclic models.

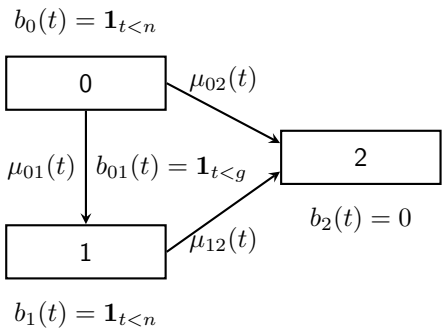
Life insurance and pension products are modeled by identifying states in a Markov model (of the life of the insured), by attaching payment intensities to the states, and by attaching lump-sum payments to the transitions. We use $b_i(t)$ to denote the payment stream in state i and use $b_{ij}(t)$ to denote the lump sum due on transition from i to j at time t.

An example product in a two-state model is the temporary life annuity, where repeated payments are made to the policy holder until some expiration date n, provided that he or she is alive. Our model contains two states: 0, in which the policy holder is alive, and 1, in which the policy holder is dead. Obviously, there is just a single transition: from 0 to 1. We have some mortality intensity $\mu_{01}(t)$, and because no payment is to occur at or following death, we know that $b_{01}(t) = 0$ and $b_1(t) = 0$. Using the syntax $\mathbf{1}_\phi$ to represent a function that returns 1 just in case ϕ holds, and 0 otherwise, we have $b_0(t) = \mathbf{1}_{t<n}$. That is, the policy pays a constant stream of one unit of currency until $t \geq n$ or until the policy holder dies. Figure 2a shows this product.

We can extend our temporary life annuity with a disability sum. A disability sum pays a lump sum when the policy holder is declared unfit to work prior



(a) Temporary life annuity



(b) Temporary life annuity with disability sum

Figure 2: Products

to some expiration g. For this task, we use a three-state model where state 0 represents active labor-market participation, state 1 represents disability, and state 2 represents death. Clearly, we have transitions from 0 to 1, from 1 to 2, and from 0 to 2. It would be possible to have a transition from 1 to 0, representing the policy holder returning to the labor market after a period of disability, but we will assume that this product pays its disability benefit at most once. Because both able-bodied and disabled holders are alive, we have $b_0(t) = b_1(t) = \mathbf{1}_{t<n}$, just as in our original product. Additionally, assuming that the disability sum is one unit of currency, we have $b_{01}(t) = \mathbf{1}_{t<g}$. Figure 2b shows the extended product.

In the AML system, we divide our product models into two components: a *risk model* consisting of the transition intensities and a *product* consisting of the payment streams and lump-sum payments. We use the term *state model* to refer to the collection of states and transitions that are available in a given Markov model, as separate from the risk model and the product. A risk model and product can be combined if they are defined within the same state model.

The *statewise reserve* $V_j(t)$ is the reserve at time t given that the insured is in state j at time t. This is the expected net present value at time t of future payments in the product, given that the insured is in state j at that time, and given information up to time t. The *principle of equivalence* states that the reserves at the beginning of the product should be 0 — that is, the expected premiums should equal the expected benefits.

We compute the statewise reserves using Thiele's differential equations (see

$$\frac{d}{dt}V_j(t) = \left(r(t) + \sum_{k;k\neq j} \mu_{jk}(t)\right) V_j(t) - \sum_{k;k\neq j} \mu_{jk}(t)V_k(t) - b_j(t) - \sum_{k;k\neq j} b_{jk}(t)\mu_{jk}(t)$$

where
$V_j(t)$ is the statewise reserve for state j at time t
$r(t)$ is the interest rate at t
$\mu_{jk}(t)$ is the transition intensity from j to k at t
$b_j(t)$ is the payment intensity in state j at t
$b_{jk}(t)$ is the lump-sum payment due on transition from j to k at t

Figure 3: Thiele's differential equations

Figure 3). While it is beyond the scope of this paper to explain Thiele's differential equations in detail, note that the parameters can be divided into three categories: those that come from a product (the $b_j(t)$ and $b_{kj}(t)$), those that come from a risk model (the $\mu_{jk}(t)$) and the interest rate, which is a property of neither. The resulting system of equations will contain one equation for each state in the product model.

In AML, a *calculation basis* for some product consists of a risk model that matches the product's state model, a model of the interest rate $r(t)$, and any additional information that might be necessary to construct the differential equations for the product.

2 Domain-specific languages

A domain-specific language (DSL) is a formal notation that is specifically tailored to a particular application area, such as digital electronic hardware design [10, 3] or financial contracts [11]. Typically, the term refers to *programming languages*: it is possible to construct programs, rather than just describe raw data. These domain-specific languages should be contrasted with general-purpose programming languages, such as C or Java, which do not provide specific, specialized support for one domain.

In general, a domain-specific language allows a domain expert (in this case, an actuary) to succinctly and precisely express concepts from the domain (in this case, life insurance and pension products). By using a domain-specific language, experts can write programs that may have been too complicated to write in a general-purpose language. Additionally, they do not need to learn the details of programming in a modern, complicated language. This enables a very short turnaround between coming up with an idea and seeing it run, enabling greater productivity and creativity through easy experimentation. Additionally, the communication overhead between the domain expert and dedicated software

developers is eliminated.

Some DSLs are defined inside of a general purpose programming language, rather than being implemented in a stand-alone manner. These languages are referred to as *embedded domain-specific languages* (EDSL). EDSLs tend to be best for skilled programmers working in one of their domains of expertise, while stand-alone DSLs have the flexibility to adapt to the needs of users who aren't professional programmers.

The Actulus Modeling Language, or AML, is a stand-alone domain-specific language. AML contains three primary components: product descriptions, computation definitions, and administrative information.

Product descriptions, described in detail in Section 3, describe insurance products and some of the associated information that is necessary for calculating reserves. There is special support in AML for preventing common mistakes in AML product definitions without having to run the code.

Computation definitions define calculations to be performed on the products and associated data that were defined using AML product descriptions. They resemble a functional programming language, which is a programming language based on mathematical notation and functions, with added safety features.

Finally, administration information supports managing the "lifecycle" of a pension product instance: creating the instance, maintaining and evolving the information associated with the instance, handling of monthly payments, production of annual statements, and so on.

3 AML product descriptions

In this section, we present an extended example of AML. We demonstrate state models, product definitions, and risk models, with a final calculation of the reserves for a product. In AML, the order of definitions is of no significance — users are free to structure their models how they wish.

3.1 A customer

We now present an AML program for computing the reserve of a whole life insurance for our favorite customer Jane who was born day one of Year 2000:

```
value jane : Person = Person("Jane", TimePoint(2000,1,1), Female)
```

The AML statement above defines the variable `jane` of type `Person` to be a record for which:

- `jane.Name` is `"Jane"`
- `jane.BirthDate` is January 1, 2000
- `jane.Gender` is `Female`

The `Person` type and the `Person(...)` operation are defined in a standard library.

3.2 A state model

As far as a whole life insurance is concerned, a customer may be alive or dead. Furthermore, a customer may die but cannot be resurrected. This corresponds to the state model in Figure 1a. Instead of numbering states, AML state model definitions name them.

```
statemodel LifeDeath(p : Person) where
  states =
    alive
    dead
  transitions =
    alive -> dead
```

The AML code above defines `LifeDeath(p)` to be a state model with two states and one transition which describes the behaviour of some person `p` from the point of view of a whole life insurance. In particular, `LifeDeath(jane)` describes the behaviour of Jane.

Inside the definition of `LifeDeath(p)` there are no references to `p`, so `p` seems superfluous. As we shall see in Section 4, however, the parameter `p` turns out to be useful for the early detection of certain errors.

3.3 A risk model

Jane dies according to the law of Gompertz-Makeham:

```
riskmodel RiskLifeDeath(p : Person) : LifeDeath(p) where
  intensities =
    alive -> dead by gompertzMakehamDeath(p)
```

The `gompertzMakehamDeath(p)` construct is defined in the standard library; it returns the mortality intensity of the person `p` as a function of time. Thus, at time `t` the mortality of `p` is `gompertzMakehamDeath(p)(t)`.

3.4 A whole life insurance

Jane's whole life insurance specifies that the insurance company has the obligation to pay an amount of money when Jane dies. For simplicity, we assume that the insurance company pays $1. A whole life insurance looks like this:

```
product WholeLifeInsurance(p : Person) : LifeDeath(p) where
  obligations =
    pay $1 when(alive -> dead)
```

In particular, `WholeLifeInsurance(jane)` describes Jane's whole life insurance.

3.5 A calculation basis

An AML calculation basis specifies the assumptions under which a reserve should be computed:

```
basis BasisLifeDeath(p : Person) : LifeDeath(p) where
  riskModel = RiskLifeDeath(p)
  interestRate = (t : TimePoint) => 0.05
  maxtime = p.BirthDate + 120
```

The calculation basis above specifies that the person `p` dies as specified by the given risk model except that no person can reach the age of 120. The interest rate specification `(t : TimePoint) => 0.05` denotes a function that, given any time `t`, returns `0.05`. In other words, the interest rate is always five percent. The three parameters `riskModel`, `interestRate` and `maxtime` are common to all calculation bases, but some will require extra information.

3.6 A reserve computation

Now define:

```
value r : Money = reserve(TimePoint(2030, 1, 1), alive,
                          WholeLifeInsurance(jane),
                          BasisLifeDeath(jane))
```

The quantity `r` above is the reserve at January 1, 2030 of Jane's whole life insurance, provided Jane is alive at that time.

To get a complete AML program one may glue the boxed AML snippets of Sections 3.1–3.6 together. Running that program through the AML system computes the reserve `r`; see Section 5 for details. Afterwards, the AML system allows users to inspect the reserve in various ways.

Figure 4 shows the statewise reserves as a function of time for the example whole life insurance.

3.7 A term insurance

Now consider a term insurance with expiration `e`:

```
product TI(p : Person, e : TimeSpan) : LifeDeath(p) where
  obligations =
    at t pay $1 when(alive -> dead)
                provided(t < p.BirthDate + e)
```

As an example, `TI(jane, TimeSpan(years=65))` specifies that the insurance company has the obligation to pay $1 when Jane dies, provided Jane is less than 65 years old at that time.

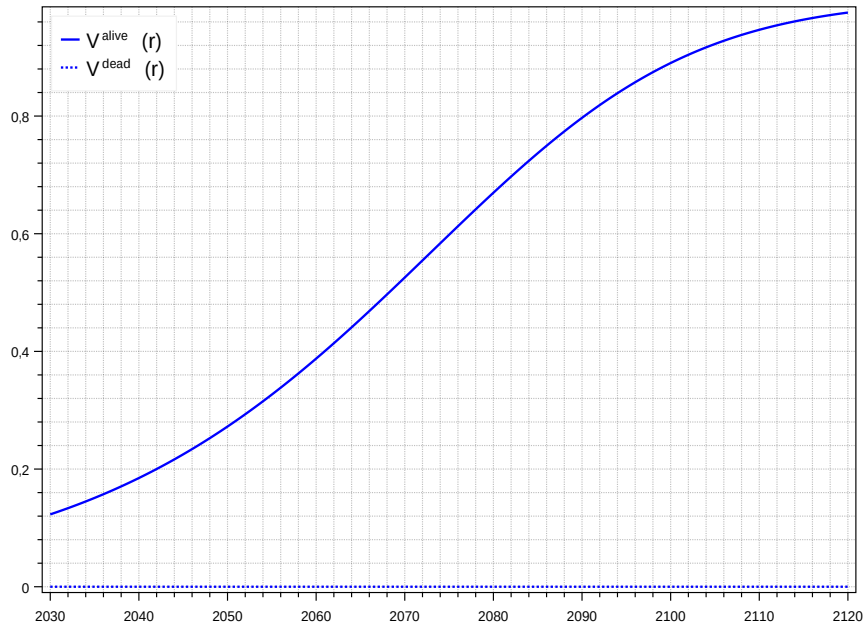


Figure 4: Statewise reserves for a whole life insurance as functions of time. The solid line represents the reserve in the `alive` state; the dotted line represents the reserve in the `dead` state.

Or, put another way, the term insurance expires at time `p.BirthDate + e` and the insurance company only pays if the time `t` is less than that. The obligation starts with `at t` which effectively says that `t` denotes time.

3.8 A life annuity

Now consider a state model for which alive customers can be disabled or active, corresponding to Figure 1b:

```
statemodel Disability(p : Person) where
  states =
    active
    disabled
    dead
  transitions =
    active -> disabled
    active -> dead
    disabled -> active
    disabled -> dead
```

The associated risk model must supply intensities for all four transitions. Having the state model above, we may express a life annuity `LA` with retirement year `r` for which disabled customers do not have to pay premiums:

```
product LA(p : Person, r : TimePoint) : Disability(p) where
  obligations =
    at t pay $1 per year provided(not dead and t >= r)
  premiums =
    at t pay $1 per year provided(active and t < r)
```

The `P per year` construct converts a payment `P` into a payment intensity. A payment intensity represents a continuous flow of money; it is a convenient approximation of payments which occur regularly.

The `P provided(C)` construct modifies a payment or payment intensity `P` according to the condition `C`. Conditions can be made up of states like `dead` and comparisons like `t >= r` using the logical connectives `and`, `or` and `not`.

3.9 A two life insurance

Now consider a state model for a couple:

```
statemodel TwoLife(insured : Person, coinsured : Person) where
  states =
    alive_alive
    alive_dead
    dead_alive
    dead_dead
  transitions =
    alive_alive -> alive_dead
    alive_alive -> dead_alive
    alive_dead -> dead_dead
    dead_alive -> dead_dead
```

As an example, `alive_dead` is the state in which the insured is alive and the co-insured is dead. In Danish insurance practice, insured and co-insured cannot die simultaneously, so the state model only has four transitions. A first to die insurance could read:

```
product FirstToDie(p : Person, q : Person) : TwoLife(p, q) where
  obligations =
    pay $1 when(alive_alive -> any)
```

The contract above pays $1 when the first among insured and co-insured dies. The `P when(E)` construct takes a payment `P` and an event `E` as parameters and returns a payment intensity.

An event can have form `t == t0` which indicates that payment should occur when the time `t` equals some value `t0`. Furthermore, an event can have form `S -> T` where `S` and `T` are state sets.

In the example above, `alive_alive` is a singleton set whose sole member is the `alive_alive` state. Furthermore, `any` is the set of all states of the state model.

A transition `S -> T` from state set `S` to state set `T` denotes the set of transitions which go from a member of `S` to a member of `T` and which furthermore is listed as a possible transition in the state model. The following three events are equivalent:

```
alive_alive -> any
alive_alive -> alive_dead or dead_alive
alive_alive -> not alive_alive
```

The AML compiler does not permit state models to contain transitions like `alive_alive -> alive_alive` from a state to the state itself.

3.10 A more complex basis

Recall the calculation basis used for Jane:

```
basis BasisLifeDeath(p : Person) : LifeDeath(p) where
  riskModel = RiskLifeDeath(p)
  interestRate = (t : TimePoint) => 0.05
  maxtime = p.BirthDate + 120
```

The calculation basis above specifies a constant five percent interest rate. An alternative could be a interest rate specified and tabulated by some Financial Services Authority (FSA):

```
value yieldcurvedata : List(Real * Real) =
  [ (0.25, 0.0040560),
    (0.5, 0.0040560),
    (1, 0.0040560),
    (2, 0.0063869), ...]
```

The calculation basis, however, needs the yield curve as a function `interestRate`, presumably interpolating the FSA yield curve data:

```
basis FsaBasisLifeDeath(p : Person) : LifeDeath(p) where
  riskModel = RiskLifeDeath(p)
  interestRate = (t : TimePoint) => fsaYieldCurve(t)
  maxtime = p.BirthDate + 120
```

In AML, the interpolation may be done by externally defined C# functions such as `fsaYieldCurve(t)`.

The example above indicates how an FSA yield curve may be used. FSA mortality data can be treated similarly.

4 Static correctness checking

The AML system is able to detect a number of errors before a program is even run. This is achieved through the use of a static type system, in many ways similar to the type systems found in languages such as Java or C#. Because AML is specifically designed for actuarial calculations, the type system has support for detecting errors related to the actuarial domain.

In a language without static types, it is possible to confuse numbers that represent different quantities. For example, it should make sense to add two spans of time, but not to add two dates. Nevertheless, both may be specified in years, months, and days. Given the definitions:

```
value birthdate : TimePoint = TimePoint(1984, 7, 12)
value millenium : TimePoint = TimePoint(2001, 1, 1)
value age : TimeSpan = TimeSpan(years=30)
value halfYear : TimeSpan = TimeSpan(months=6)
```

AML will accept operations such as:

```
age + halfYear        // yielding a TimeSpan(years=30, months=6)
birthdate + halfYear        // yielding a TimePoint(1985, 1, 12)
millenium - birthdate                        // yielding a TimeSpan
```

yet it will reject meaningless operations such as:

```
birthdate + millenium
```

If mere numbers were used to represent dates and times, then it might be possible to confuse time points and time spans.

In addition to these simple errors, which can be caught in almost any statically-typed programming language, AML can catch errors that are specific to the actuarial domain. A simple example of such an error involves using the wrong state model. In this erroneous version of `DisabilityInsurance`, a state model is used that lacks the `disabled` state:

```
product DisabilityInsurance(p : Person) : LifeDeath(p) where
  obligations = pay $1 provided(disabled)
```

This error is detected before the program is even run.

Additionally, AML is able to check the correspondence between products and calculation bases. A calculation basis is compatible with a product if it

can provide all of the information that is necessary to perform computations using the product. At a minimum, it must provide an interest rate model and transition intensities for each transition in the product's state model. Some products, however, require more information.

The product `SpouseBenefits` provides a one-time payment at the death of the insured, but only if he or she is married. The marital status of the insured is first checked upon his or her death — a deathbed marriage entitles the surviving spouse to full benefits. Thus, our evaluation of the product depends on having a model for whether the insured is married.

```
product SpouseBenefits(p : Person) : LifeDeath(p) where
  obligations = at t pay $1
                when(alive -> dead)
                provided(married)
                given(married ~ basis.marriageProb(p, t))
```

The `P given(V ~ D)` construct states that inside `P`, the variable `V` is distributed according to the distribution `D`. Above, `married` is a Boolean variable which is true with a probability prescribed by `basis.marriageProb(p, t)`.

The dot-notation means that `marriageProb` must be defined in the calculation basis. AML examines products and calculation bases, and checks that each of a product's basis variables exists and that it has the correct type in the applied basis. An applicable basis could read

```
basis SpouseBasis(p : Person) where
  riskModel = RiskLifeDeath(p)
  interestRate = (t : TimePoint) => 0.5
  maxtime = p.BirthDate + 120
  marriageProb = marriage
```

where

```
function marriage(p : Person, t : TimePoint) : Dist(Bool) =
  boolDist(if p.Gender == Male then 0.8 else 0.55)
```

The definition above makes the approximation that males and females are married with probability 0.8 and 0.55, respectively, independent of age. The `boolDist` library function converts a probability to a distribution. A basis like

```
basis ThisIsNotASpouseBasis(p : Person) where
  riskModel = RiskLifeDeath(p)
  interestRate = (t : TimePoint) => 0.5
  maxtime = p.BirthDate + 120
```

cannot be used with `SpouseBenefits`, as it contains no model of marriage probabilities.

AML's type system is able to check types that include AML values. This means, for example, that the system can use the `Person` parameter to a product or basis to ensure that a product whose payments are calculated based on a 20 year old woman is not accidentally calculated in a basis for a 70 year old man. This is why state models often take the person whose life they measure as a parameter. In order to use a product with a calculation basis, they must be defined in the same state model, *with all parameters equal*. Including the person as a parameter to the state model ensures that it must match.

Thus, the AML compiler would protest against

```
value r : Money = reserve(2030, "alive",
                          WholeLifeInsurance(jane),
                          BasisLifeDeath(John))
```

because the `reserve` function requires the product and the calculation basis to be built upon the *same* state model. Above, however, the product builds upon `LifeDeath(jane)` whereas the calculation basis builds upon `LifeDeath(John)`. Because the type system is aware of the *values* `John` and `jane`, it can check whether they are equal.

5 Computing reserves

One use of AML product descriptions, risk models and calculation bases is to compute the reserve, that is, the expected present value of the future payments to be made in a product. Here we consider an example and describe how such reserves are computed in AML.

5.1 Example reserve calculation

The AML `reserve` function call shown in Section 3.6 computes the reserve of the example product developed in Sections 3.1 through 3.5. Concretely, the AML implementation generates and then numerically solves this specialized instance of Thiele's differential equations from Figure 3:

$$\begin{aligned}\frac{d}{dt}V_{\text{alive}}(t) &= r(t)V_{\text{alive}}(t) - b_{\text{alive}}(t) \\ &\quad - \mu_{\text{alive,dead}}(t)\left(b_{\text{alive,dead}}(t) + V_{\text{dead}}(t) - V_{\text{alive}}(t)\right) \\ \frac{d}{dt}V_{\text{dead}}(t) &= r(t)V_{\text{dead}}(t) - b_{\text{dead}}(t)\end{aligned}$$

where

$V_{\text{alive}}(t)$ is the reserve at time t if Jane is alive (and similar for $r_{\text{dead}}(t)$).

$r(t)$ is the interest rate at t. From our product definition, we have that $r =$ `BasisLifeDeath(jane).interestRate`, so $r(t) = 0.05$ for all t. The interest rate may depend on time, but it does not in this case.

$b_{\text{alive}}(t)$ is the payment intensity (in dollars per year) at time t provided Jane is alive. Since the whole life insurance product does not specify any payments while Jane is alive we have $b_{\text{alive}}(t) = 0$. Likewise, $b_{\text{dead}}(t) = 0$.

$\mu_{\text{alive,dead}}(t)$ is Jane's mortality intensity at time t. We have $\mu_{\text{alive,dead}}(t) =$ `BasisLifeDeath(jane).riskModel(alive->dead)(t) = gompertzMakehamDeath(jane)(t)`.

$b_{\text{alive,dead}}(t)$ is the payment upon death at time t, that is, \$1 as specified by the product.

As initial conditions, the differential equation solver employed by the AML reserve function takes

$$\begin{aligned} V_{\text{alive}}(t_{\max}) &= 0 \\ V_{\text{dead}}(t_{\max}) &= 0 \end{aligned}$$

where $t_{\max}$ is `BasisLifeDeath(jane).maxtime` which is `jane.BirthDate+120` (Jane's 120 year birthday).

The function call `reserve(2030, "alive", ...)` in Section 3.6 solves this specialized instance of Thiele's equations right to left and returns $V_{\text{alive}}(2030)$.

5.2 Implementation of reserve computations

Reserves are calculated by numerical solution of Thiele's differential equations, so that all risk models can be handled, even those containing cycles, for instance because of reactivation (Section 6.1).

First, the relevant instance of Thiele's differential equations are generated from the AML specifications of product, risk model and calculation basis.

Then, the generated equations are solved. This can be done in a number of ways:

- By interpreting a representation of the equations in a general ODE solver, such as a fixed-step Runge-Kutta 4th order solver on a standard CPU. This is done in the current AML prototype calculation kernel.
- By interpreting a representation of the equations in a general solver as above, but using a farm of standard CPUs, for instance via Amazon's Elastic Compute Cloud. We have successfully experimented with this approach.
- By compiling the equations into CUDA C code [14] that represents a solver specialized to those particular equations. This CUDA C code can then be executed on general-purpose graphics processors (GPGPUs), which

may have 200–1500 simple parallel processors and may achieve 100-fold speedup over CPU-based solvers. Extensive experiments in this direction have been conducted; see further below.

- By a combination of the above approaches, such as generating specialized CUDA C code and running it on GPGPU-equipped systems in the cloud. We have not yet attempted this.

This range of possible solutions highlights one advantage of using AML: the author of the product, risk model and calculation basis specifications need not care what technology is subsequently used to compute the reserves. Conversely, a change in underlying technology, for instance to use adaptive-step solvers or a new generation of GPGPUs, does not require any change in the AML specifications at all.

Also note that because of the automated pipeline from AML product description and so on to numeric solution, an actuary may experiment with product definitions, risk models and calculation bases and have the reserve (and similar quantities) recalculated immediately without tedious low-level programming in C#, Java, C, C++, or similar general-purpose languages.

A companion ICA 2014 paper [4] describes how to automatically generate numeric solvers for Thiele's differential equations from given AML product, risk model and calculation basis specifications. That paper also describes how the generated code should be structured for very high performance on GPGPUs. Comprehensive experiments have been conducted with this approach, using Runge-Kutta 4th order solvers which seem particularly amenable to execution on current (2013) hardware. The differential equations corresponding to a portfolio of products can be solved in parallel with high efficiency.

6 Computation examples

The AML system provides built in support for computing quantitative aspects of products defined in the language, such as reserves, cashflow, transition probabilities, etc. We give examples of computations based on reserves and illustrate how one can compute reserves in different scenarios, stress the computations, and post process quantities.

6.1 Disability product with reactivation

Our computational model is based on solving differential equations numerically. As a consequence, it is straightforward to compute reserves for products involving cycles in the underlying state and risk models; for example, disability products with reactivation. This is decidedly non-trivial in computational models based on analytical solutions.

Recall the `Disability` state model in Section 3.8:

```
statemodel Disability(p : Person) where
  states =
    active
    disabled
    dead
  transitions =
    active -> disabled
    active -> dead
    disabled -> active
    disabled -> dead
```

and note that the model allows not only a transition from `active` to `disabled`, but also the *reactivation* transition from `disabled` to `active`.

As an example involving reactivation, we consider a disability annuity `DA`, where the insurance company pays an annuity during disability until expiry `e`:

```
product DA(p : Person, e : TimePoint) : Disability(p) where
  obligations =
    at t pay $1 per year provided(t < e and disabled)
```

The risk model of the calculation basis allows us to include or exclude reactivation:

```
riskmodel RiskDisability(p : Person, r : Bool) : Disability(p)
  where
    intensities =
      active   -> disabled by gompertzMakehamDisability(p)
      active   -> dead     by gompertzMakehamDeath(p)
      disabled -> active   by
        if r then reactivation(p) else (t : TimePoint) => 0.0
      disabled -> dead     by gompertzMakehamDeath(p)
```

where `r` is a boolean controlling whether to include reactivation. Reactivation is excluded by specifying an intensity of zero. For example purposes, we choose a very simple intensity function when reactivation is included:

```
function reactivation(p : Person) : TimePoint -> Real =
  (t : TimePoint) => 10 ^ (-0.03 * (t - p.BirthDate))
```

Note that the return type of the function `reactivation` contains an arrow. Arrows in types denote functions — in this case, `reactivation` takes a person as its argument and returns a new function from points in time to the real numbers.

Define a basis `BasisDisability` for the risk model, as previously, which propagates reactivation inclusion/exclusion to the risk model:

```
basis BasisDisability(p : Person, r : Bool) : Disability(p)
  where
    riskModel = RiskDisability(p, r)
    interestRate = (t : TimePoint) => 0.05
    maxtime = p.BirthDate + 120
```

We can now easily compare the reserve for the product in a model with reactivation, `r0`, to the reserve for the product without, `r1`:

```
value r0 : Reserve =
  reserve(2030, DisabilityAnnuity(jane, 2065),
          BasisDisability(jane, true))
value r1 : Reserve =
  reserve(2030, DisabilityAnnuity(jane, 2065),
          BasisDisability(jane, false))
```

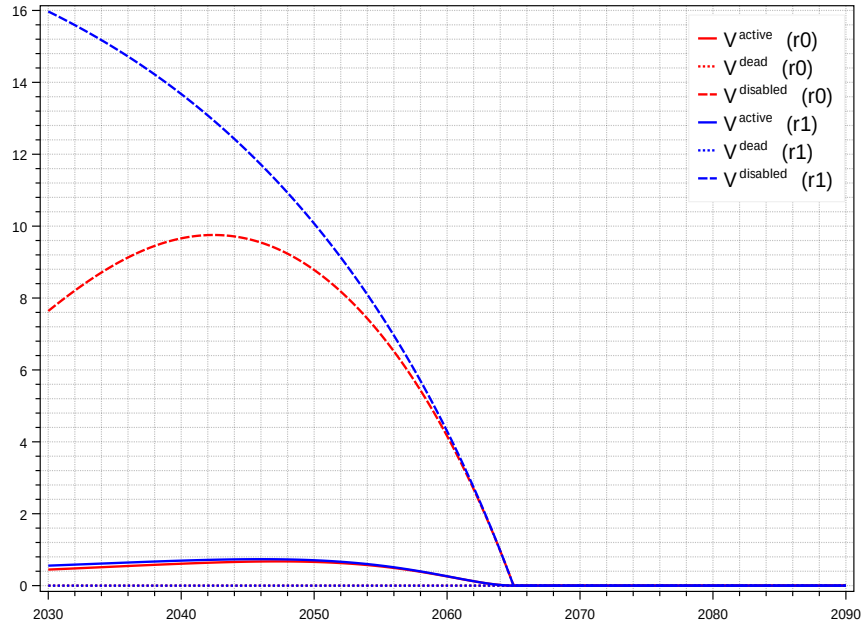


Figure 5: Reserves for a disability annuity with and without reactivation. The red reserves are computed for a model with reactivation; the blue reserves for model without.

Figure 5 shows the results. With a positive reactivation intensity, the reserve in the computation with reactivation is smaller than the reserve in the computation

without, because the product only pays in the `disabled` state. The example reactivation intensity tends towards zero, and thus the reserve in the `disabled` state with reactivation tends towards the reserve without reactivation.

6.2 Solvency II stress

The EU Solvency II legislation specifies a number of stresses to be used when computing the solvency capital requirements for an insurance company. We give examples of implementing such stresses directly in AML for a product instance.

The Solvency Capital Requirements (SCR) Life underwriting risk module [6] specifies a number of stresses to the intensities of the risk model. Define a stressed risk model `StressedRiskLifeDeath` as:

```
riskmodel StressedRiskLifeDeath
  (p : Person,
   s_mu : (TimePoint -> Real) -> (TimePoint -> Real))
                                          : LifeDeath(p)
  where
    intensities = alive -> dead by s_mu(gompertzMakehamDeath(p))
```

The syntax `s_mu : (TimePoint -> Real) -> (TimePoint -> Real)` declares that `s_mu` is a function which transforms the original mortality intensity function which has type `TimePoint -> Real` (in this case `gompertzMakehamDeath`) to new stressed intensity function with that same type. For example, the $\text{Life}_{\text{mort}}$ mortality risk specifies a permanent 15% increase in mortality rates:

```
function LifeMort(mu : TimePoint -> Real) : TimePoint -> Real =
  (t: TimePoint) => 1.15 * mu(t)
```

The function `LifeMort` applies the mortality stress to an existing intensity function `mu`, resulting in a new intensity. Giving the identity function for the stress `s_mu` parameter in `StressedRiskLifeDeath` results in a risk model with no stress.

Next define a basis `StressedBasisLifeDeath` which propagates the stress to the risk model:

```
basis StressedBasisLifeDeath
  (p : Person,
   s_mu : (TimePoint -> Real) -> (TimePoint -> Real))
                                          : LifeDeath(p)
  where
    riskModel = StressedRiskLifeDeath(p, s_mu)
    interestRate = (t : TimePoint) => 0.05
    maxtime = p.BirthDate + 120
```

As above, it is straightforward to compute reserves with and without the stress, and compare the results. Figure 6 shows the effect of the mortality stress for a

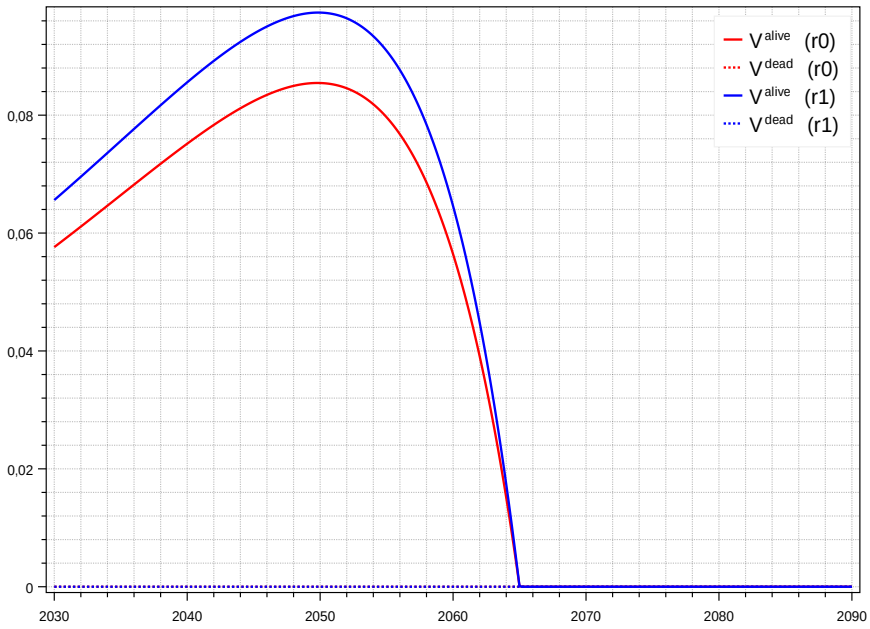


Figure 6: Reserves for a term insurance. The red reserves (`r0`) are computed without stress; the blue reserves (`r1`) with a mortality stress by a permanent 15% increase in mortality rates.

term insurance product. As expected increasing mortality rates leads to larger reserves since the product pays a sum upon death of the insured.

We may further extend `StressesBasisLifeDeath` to also include a stress to the interest rate:

```
basis StressedBasisLifeDeath
  (p : Person,
   s_mu : (TimePoint -> Real) -> (TimePoint -> Real)
   s_int : (TimePoint -> Real) -> (TimePoint -> Real))
                                          : LifeDeath(p)
  where
    riskModel = StressedRiskLifeDeath(p, s_mu)
    interestRate = s_int(r)
    maxtime = p.BirthDate + 120
```

where `r` is the original interest rate, as a function of time.

For example, the Solvency Capital Requirements (SCR) Market risk module specifies an interest rate risk. This stress is specified as a table of relative changes, both up and down, for maturities ranging from 0.25 to 30 years. It is possible to express the table in AML and to use it to stress the interest rate.

6.3 Duration

As a final example, we show how computed reserves can be post-processed to compute the sensitivity of a product to changes in the interest rate (see for example Fabozzi [7, Chapter 4]).

The *duration* D is computed as

$$D = \frac{V[r + \Delta r] - V[r - \Delta r]}{2\Delta r}$$

and the *modified duration* as

$$ModD = \frac{D}{V[r]}$$

where $V[r]$ denotes the computed reserve for the product in question with an interest rate of r.

To express such post processing in AML we start by parametrizing our calculation basis with the interest rate:

```
basis LifeDeathBasis(p : Person, r : TimePoint -> Real)
                                                  : LifeDeath(p)
  where
    riskModel = RiskLifeDeath(p)
    interestRate = r
    maxtime = p.BirthDate + 120
```

It is then straightforward to compute the quantities in the equations for the duration and the modified duration:

```
function interest(t : TimePoint) : Real = 0.05

function shift(r : TimePoint -> Real,
               s : Real) : TimePoint -> Real =
  (t : TimePoint) => r(t) + s

value V0 : Money =
  reserve(2030, "alive", P, LifeDeathBasis(jane, interest))

value Vplus : Money =
  reserve(2030, "alive", P,
          LifeDeathBasis(jane, shift(interest, 0.001)))

value Vminus : Money =
  reserve(2030, "alive", P,
          LifeDeathBasis(jane, shift(interest, -0.0001)))

value D : Real = (Vplus - Vminus) / (2 * 0.001)
```

```
value ModD : Real = D / V0
```

where P is the product instance, for example

```
value P : Product = LA(jane, 35)
```

7 Future work

Future work includes the design of a more general calculation language AML Computation, for describing interest rate stress, mortality stresses, and so on, and possibly for aggregating the results of computations and simulations. Also, we are actively pursuing a range of approaches to scalable high-performance computation on AML product portfolios [4]. Finally, we will consider whether to develop an actual domain-specific language AML Administration for the management of life insurance contracts (creation, monthly updates, annual reports, and so on), or whether a more classical library-style application programmers interface (API) would serve the various use contexts for AML better.

8 Related work

To the best of our knowledge, there are no previous proposals for domain-specific languages within the life insurance and pension domain. However, there are several well-known domain-specific languages in related areas, in particular finance; we give a brief overview here.

Cash Flow Reengineering Perhaps the earliest financial DSL in the literature is Risla, described in Arnold et al. [2]. Risla was used to create new financial products by composing already-existing cash flows, providing a high-level specification that was accessible to experts. COBOL code could then be generated from the Risla specification. This code integrated the new product into the business processes of Bank MeesPierson and CAP Volmac, the language's developers. In 2003, Mogensen [12] presented a language used for much the same purpose as Risla. The most interesting feature of this language is its ability to ensure that payment streams are not used twice, which can be non-trivial to verify in complex products. The resulting language was used by an unnamed major Danish bank, which had originally approached Mogensen's institute for help because their prior practice of implementing pricing code by hand in a general purpose language was unsatisfactory, and they concluded that the custom language was helpful.

Financial Contracts Jones et al. present a language for building complex financial contracts by combining simple contracts [11]. This language is an EDSL in the programming language Haskell. This work has been expanded upon

throughout the years by Andersen et al. [1], Frankau et al. [9] and Flænø Werk et al. [8].

Controlled Natural Languages Pace and Rosner [13] have an alternate approach for formulating a DSL for financial products. Instead of following the conventions of programming languages, they provide an outline of a controlled natural language — a subset of a natural language with a precise formal semantics — that can be used to describe financial contracts. This language is based on a formalized deontic logic for describing these contracts.

Legal Specifications In 2010, the United States' Securities and Exchange Commission released a recommendation [15, pp. 213-214] that investors represent certain kinds of financial instruments using Python code. The response of ACM SIGPLAN (the leading professional organization for programming language experts) [5] agrees that a formal, algorithmic specification of these financial products and their analysis is worthwhile, though they recommend a functional language such as F# or a domain-specific language or application library as being more suited to the problem.

9 Conclusion

We have described a domain-specific language, or notation, for use by actuaries when designing and evaluating life insurance and pension products. A product description in this language can be used for a range of purposes within a life insurance company, thus ensuring coherence between product administration, reporting, solvency computations and so on.

We also described a general calculation kernel that can compute reserves and other quantities of interest, given product descriptions expressed in the domain-specific language.

The separation into a specific product description and a general calculation kernel is expected to bring long-term benefits. For instance, changes to the underlying technological platform will affect the calculation kernel only, not the product descriptions; these remain isolated from the technological change that will inevitably happen over a pension product's multidecade lifespan.

References

[1] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):485–516, November 2006.

[2] B. R. T. Arnold, Arie van Deursen, and Martijn Res. Algebraic specification of a language for describing financial products. Technical report, Eindhoven University of Technology, 1995.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 174–184. ACM, September 1998.

[4] David Christiansen, Nicolai Dahl Blicher-Petersen, Christian Harrington, Lars Henriksen, and Peter Sestoft. High-performance reserve calculations for life insurance portfolios. In *ICA 2014*, 2014.

[5] ACM US Public Policy Council. Comment letter on file no. s7-08-10. `http://www.sec.gov/comments/s7-08-10/s70810-89.pdf`, August 2010. Accessed December 16, 2011.

[6] European Insurance and Occupational Pensions Authoriy. Quantitative Impact Study 5 (QIS5): Technical specifications. `https://eiopa.europa.eu/consultations/qis/insurance/quantitative-impact-study-5/index.html`, July 2010. Accessed April 5, 2013.

[7] Frank J. Fabozzi. *Duration, Convexity, and Other Bond Risk Measures*. John Wiley and Sons, 1999.

[8] Michael Flænø Werk, Joakim Ahnfelt-Rønne, and Ken Friis Larsen. An embedded DSL for stochastic processes. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12, pages 93–102, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1577-7.

[9] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(01):27–45, 2009.

[10] Geraint Jones and Mary Sheeran. Relations and refinement in circuit design. In *Proceedings of the BCS-FACS Workshop on Refinement, Workshops in Computing*, pages 133–152. Springer-Verlag, 1991.

[11] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 280–292, New York, NY, USA, 2000. ACM.

[12] Torben Mogensen. Linear Types for Cashflow Reengineering. In Manfred Broy and Alexandre Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, chapter 2, pages 823–845. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2003.

[13] Gordon Pace and Michael Rosner. A Controlled Language for the Specification of Contracts. In Norbert Fuchs, editor, *Controlled Natural Language*, volume 5972 of *Lecture Notes in Computer Science*, chapter 14, pages 226–245. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14417-2.

[14] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[15] United States Securities and Exchange Commission. Proposed rule: Asset-backed securities. `http://www.sec.gov/rules/proposed/2010/33-9117.pdf`, 2010. Accessed December 16, 2011.