Towards a software system for vulnerability assessment

Daniel Lincke

November 6th, 2007
Outline

The concept vulnerability

Why formalisation?

A formal framework of vulnerability

A modell in Haskell

Toward a C++ library
The concept of vulnerability

- Vulnerability is a central concept in climate change research (as well as in a number of other research contexts)
- ”Bangladesh is vulnerable to sea level rise”
- IPCC: Working Group II: ”Impacts, adaptation and vulnerability”
- At PIK: Research Domain II: ”Climate Impacts and Vulnerabilities”
Need for a common understanding

- the term is conceptualized in many different ways by the various scientific communities
- but also within the climate change scientific community, there is confusion regarding the notion of vulnerability
- Research has identified a need for formalisation and operationalisation.
- IPCC definition:
  "Vulnerability is the degree to which a system is susceptible to, and unable to cope with, adverse effects of climate change, including climate variability and extremes. Vulnerability is a function of the character, magnitude, and rate of climate change and variation to which a system is exposed, its sensitivity, and its adaptive capacity"
This is one of a multitude of definitions.
Goals of formalisation

Ionescu et al. [2005] point out four main goals:

- to ensure that examining, interpreting and representing vulnerability is done systematically, limiting the potential for analytical inconsistencies
- improve the clarity of communication of methods and results of vulnerability assessments, avoiding misunderstandings
- help users of assessment results to detect and resolve any inaccuracies and omissions
- A formal framework is a precondition for any computational approaches to assessing vulnerability and allows modellers to take advantage of relevant methods in applied mathematics, such as systems theory and game theory
Language of formalisation: mathematics

- "Mathematical notation appears as a sort of language, *une langue bien faite*, a language well adapted to its purpose, concise and precise, with rules which, unlike the rules of ordinary grammar, suffer no exception."

- consistency

- natural language may contain residual ambiguities

- a mathematical description easily translates into programming languages
The formalisation process

1. Grammatical investigation:
   - Oxford Dictionary of English: vulnerable [Soanes and Stevenson, 2003] - exposed to the possibility of being attacked or harmed, either physically or emotionally: we were in a vulnerable position | small fish are vulnerable to predators.

2. Inherent structure:
   - Vulnerability is a relative concept: it is the vulnerability of an entity to a specific evolution of its situation and with respect to a certain measure of ”well being” of the entity

3. Identifying three primitives:
   - entity, possible future evolutions, harm

4. Mathematical representation
Mathematical representation

1. entity and evolution: dynamical system
   - set of states $S$
   - transition function $\text{poss}: S \rightarrow \mathcal{P}S$
   - $s \in S$ "current situation"
   - $\text{poss}(s) = A \subseteq S$ "possible evolutions"

2. harm: measure
   - simple case: predicate impacts
     $\text{impacts}(s') \in \{0, 1\}$ for all $s' \in A$
   - degrees of harm: measure impacts
     $\text{impacts}(s') \in [0, 1]$ for all $s' \in A$
1. **Vulnerability**
   - vulnerability: a measure of the possible future harm of the entity in a given state
   - \( \text{vulnerability}(s) = \text{measure}(\{\text{impacts}(s') | s' \in \text{poss}(s)\}) \)
   - properties of the measure to be specified: e.g. monotonicity

2. **Vulnerability to a certain stimulus**
   - factor’s contribution to the transition
     \( \text{cont}(s', f) \in [0, 1] \) for all \( s' \in A \)
   - \( \text{vulnerability}_{\text{to}}(s, f) \)
     \( = \text{measure}(\{(\text{impacts}(s'), \text{cont}(s', f)) | s' \in \text{poss}(s)\}) \)
Why a modell in Haskell?

- a notation for the specification of vulnerability
- implementation of these specifications can be executed
- test and study algorithms
- avoid errors by detecting them in this modelling stage instead of the stage of implementation
- reduce costs of software development
Haskell representation: Dynamical system

- represented by a coalgebra \( f : X \rightarrow FX \) where \( F \) is a functor (Rutten [2000])

- we consider only the case here, where \( F \) is a monad

- newtype (Monad m) => System m a = System (a -> m a)

- A system can be applied:
  apply_sys (Monad m) => System m a -> m a -> m a
  apply_sys sys x = x >>= sys

- A system can be iterated:
  iterate ::(Monad m) => System m a -> m a -> Int -> m a
  iterate sys mx 0 = mx
  iterate sys mx (n+1)= apply_sys sys (iterate sys mx n)
Examples for systems:

- Monad \( m = \text{Id} \)  \( \Rightarrow \) Deterministic System
- Monad \( m = \text{P} \)  \( \Rightarrow \) Nondeterministic System
- Monad \( m = \text{SP} \)  \( \Rightarrow \) Stochastic System
- Monad \( m = \text{Fuz} \)  \( \Rightarrow \) Fuzzy System
Haskell representation: harm and possible

- poss :: (Monad m) => System m a -> m a -> [a]
- poss_t :: (Monad m) => System m a -> m a -> Int -> [[a]]
- harm :: (Monad m) => m a -> m a -> Bool
- some :: (Monad m) => (m a -> m a -> Bool) -> (System m a -> m a -> [a]) -> System m a -> m a -> Bool
- some_t :: (Monad m) => (m a -> m a -> Bool) -> (System m a -> m a -> Int -> [a]) -> System m a -> m a -> Int -> Bool
Haskell representation: vulnerability

- `simple_vulnerability ::`
  (Monad m) => System m a -> m a -> Bool
- `simple_vulnerability sys mx =`
  some harm poss sys mx
- `transitional_vulnerability ::`
  (Monad m) => System m a -> m a -> Int -> Bool
- `transitional_vulnerability sys mx n =`
  some_t harm poss_t sys mx n
Why implementation in C++?

- programs can be implemented on different levels of abstraction, reaching from machine-oriented programming to object-oriented programming (Stroustrup [1998]).
- multiparadigm programming language (McNamara [2004]).
- many compilers available, produce optimized and fast code.
- many software libraries are written in C++ (Botta et al. [2006])
- supports generic programming, vulnerability assessment is a generic task
- want to use Haskell as a high level modelling and C++ as implementation language. Approaches for an automated mapping of Haskell specification in C++ constructs exist (Zalewski et al. [2007]).
C++ implementation: basic framework

- how to implement a basic framework of category theory?
- elementary notion: function
- four ways of implementing the concept of a function
  1. dynamic polymorphism
  2. wrapper around function pointer
  3. static polymorphism using static interfaces (McNamara and Smaragdakis [2000])
  4. static polymorphism using concepts (Reis and Stroustrup [2006])
Dynamic polymorphism

template<class S, class T>
class Function {
    public:
        typedef S domain_type;
        typedef T codomain_type;
        virtual codomain_type operator() (const domain_type& s) const = 0;
};

class Test_Fun : public Function<int,int> {
    public:
        int operator() (const int& t) const
        {
            return (t*2);
        }
};
Function pointer

```cpp
template<class S, class T>
class Function {
    public:
        typedef S domain_type;
        typedef T codomain_type;
        typedef T (*pt2Func)(S const&);

        Function (pt2Func f) { my_f=f; }
        codomain_type operator () (domain_type const& x) const
            { return my_f(x); }

    private:
        pt2Func my_f;
    };

int test_fun(const int& t) return (t*2);
Function<int,int> f(test_fun);
```

Daniel Lincke, Cezar Ionescu, Jochen Hinkel, Sandy Bisaro, Sarah Wolf
Potsdam Institute for Climate Impact Research
Towards a software system for vulnerability assessment
The concept vulnerability
Why formalisation?
A formal framework of vulnerability
A modell in Haskell
Toward a C++ library
References

Static polymorphism - static interfaces

template<class F>
class Function_Interface {
  public:
    MAKE_TRAITS;
    template<class Self> static void check_structural() {
      typedef typename F::domain_type domain_type;
      typename F::codomain_type (Self::*x) (const typename F::domain_type&) const
      = &Self::operator();
      (void) x;
    }

    ...}
};

class Fun2 : public Function_Interface<Fun2> {
  ...
};

Daniel Lincke, Cezar Ionescu, Jochen Hinkel, Sandy Bisaro, Sarah Wolf
Potsdam Institute for Climate Impact Research
Towards a software system for vulnerability assessment
Static polymorphism - concepts

class Test_Fun2 {
    
    int operator() (const int& t) const { return (t*2); }
};

concept_map Function_Concept<Test_Fun2> {
    typedef Test_Fun2::domain_type domain_type;
    typedef Test_Fun2::codomain_type codomain_type;
}
Performance comparison

- to test performance we call a test procedure which gets an function and a domain_type as arguments
- this procedure is called $10^9$ times and the time is measured
- for example:
  ```cpp
template <class A, class B>
B call1 (const Function<A,B>& f, const A& x) {
  return f(x); }
```
- or:
  ```cpp
template <class F>
#ifdef __GXX_CONCEPTS__
requires Function_Concept< F>
#endif
typename F::codomain_type call4 (const F& f, const
typename F::domain_type& x) {
  return f(x); }
```
Performance comparison

- results:

<table>
<thead>
<tr>
<th></th>
<th>no optimization</th>
<th>highest optimization level (-O4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>call1, f1</td>
<td>1.52 seconds</td>
<td>1.21 seconds</td>
</tr>
<tr>
<td>call2, f2</td>
<td>1.47 seconds</td>
<td>0.96 seconds</td>
</tr>
<tr>
<td>call3, f3</td>
<td>0.94 seconds</td>
<td>0.11 seconds</td>
</tr>
<tr>
<td>call4, f4</td>
<td>0.94 seconds</td>
<td>0.11 seconds</td>
</tr>
</tbody>
</table>

- call1: dynamic polymorphism, call2: function pointer, call3: static interfaces, call4: concepts
- f1 - f4 represent the same computation
- test performed under gcc 4.0.2, SuSe Linux 10.0 (kernel 2.6.13-15.13) and on a Intel Mobile Pentium III with 866 MHz
Conclusion of the comparison

- dynamic polymorphism: standard approach, poor performance in terms of runtime
- function pointer wrappers: standard approach, poor performance in terms of runtime
- static interfaces: very fast, old fashioned, problems in terms of portability
- concepts: (might be) the future standard used in generic programming, very fast, still in the stage of prototypes

We decided to follow the concept based approach.
Concepts for basic structures

- function and function composition
- type constructor and functor
- coalgebra and coalgebra composition
- monad (?)
Function and function composition

template <class G, class F>
#ifdef __GXX_CONCEPTS__
    requires Function<F>, Function<G>,
    Convertible<F::codomain_type,G::domain_type> ...
#endif
class Function_Composition {
...
    typedef typename F::domain_type domain_type;
    typedef typename G::codomain_type codomain_type;
    
typename G::codomain_type operator()(
        const typename F::domain_type& s) const {
        return my_g( my_f (s) );
    }

private:
    F my_f; G my_g;
};
Function and function composition

```cpp
#ifdef __GXX_CONCEPTS__
template <class G, class F>
concept_map Function<Function_Composition<G, F> > { 
  ...
}
#endif
```
Function and function composition

```cpp
template<typename T> struct Identity {
    typedef T domain_type;
    typedef T codomain_type;

    codomain_type operator () (domain_type const& s) const {
        return s;
    }
};

template<typename T>
concept map Function<Identity<T> > { ... }
```
Type constructors

Computes one type out of another type:

```cpp
concept TypeConstructor<class TC> {
    typename T;
    typename FT;
};
```

Example:

```cpp
template<typename Type>
struct SP_TypeConstructor {
    typedef Type T;
    typedef SimpleProbability<Type> FT;
};
```

```cpp
template<typename Type>
concept map TypeConstructor<SP_TypeConstructor<Type> > {
    ...
}
```
Functors

class Functor f where
    fmap :: (a -> b) -> (f a -> f b)

concept Functor<class F> : TypeConstructor<F> {
    Function F_type;
    Function FMap_type;
    
    FMap_type F::FMap (F_type const&);
};

▶ why is the function type F_type part of the concept?
▶ no better solution found (technical reasons)
▶ nested templates not supported yet.
▶ axioms? FMap id = id , FMap (f . g) = (FMap f . FMap g)
Functor example

```cpp
template <typename F> 
requires Function<F> 
struct SP_FUNCTION_MAPPING {
    typedef SimpleProbability<F::domain_type> domain_type;
    typedef SimpleProbability<F::codomain_type> codomain_t;

    codomain_type operator() (domain_type const& x) const {
        F my_f; codomain_type ret;
        for (... it=x.begin(); it!=x.end(); ++it)
            ret.Insert(my_f(it->first),it->second);
        return ret;
    }
};
```

Daniel Lincke, Cezar Ionescu, Jochen Hinkel, Sandy Bisaro, Sarah Wolf
Towards a software system for vulnerability assessment
Functor example

template <typename F>
concept_map Function<SP_Function_Mapping<F> > { ... }

template<class Type, class Fun>
requires Function<Fun>
struct SP_Functor : SP_TypeConstructor<Type> {
    typedef Fun F_type;
    typedef SP_Function_Mapping<Fun> FMap_type;
    FMap_type FMap (F_type const& f) { ... } }

template<class Ty, class Fun>
concept_map Functor<SP_Functor<Ty,Fun> > { }
Coalgebra

class Coalgebra<
    class Coalg,
    : Function<Coalg>
    {
        TypeConstructor functor_type;
        typename domain_type = functor_type::T;
        typename codomain_type = functor_type::FT;
    }

Example:

    template<class T>
    struct Wrap_set {
        typedef P_TypeConstructor<T> functor_type;
        codomain_type operator() (domain_type const& x) ...
    }

Why not require functor_type to be a Functor? Second template parameter of, e.g., P_Functor is not known ...
Outlook

- still to implement: monad
- starting from that: implement the concept of a dynamical system
- concepts for measures, indicators
- generic components for vulnerability assessment
Further development

- formal framework is under continuous development
- some concepts of vulnerability are not considered yet: sensitivity, adaptive capacity
- dynamical system: include a time abstraction
  \( \phi : T \to Hom(X, FX) \) (Monoid morphism),
  newtype (Monoid t Monad m) =>
  DynSys t m x = DynSys (t -> x -> mx)
- climate change: special types of systems are considered - SES
- ...
Conclusion

- a formal framework of vulnerability is a good idea
- a system for computational vulnerability assessment is useful, but doesn’t exist
- computational vulnerability assessment is a generic task, taking advantage of existing models
- therefore we need a generic set of software components supporting these assessments
- a underlying framework of algebraic concepts supports the development of this software
- concept based programming is used
- a lot of research is to be done to get a clear, useable and consistent concept based library of algebraic constructs and of vulnerability tools
References


