Software languages for data exchange systems

L. Thomas van Binsbergen[1]

[1]Centrum Wiskunde & Informatica
l.t.van.binsbergen@cwi.nl

April 2020

Section 1

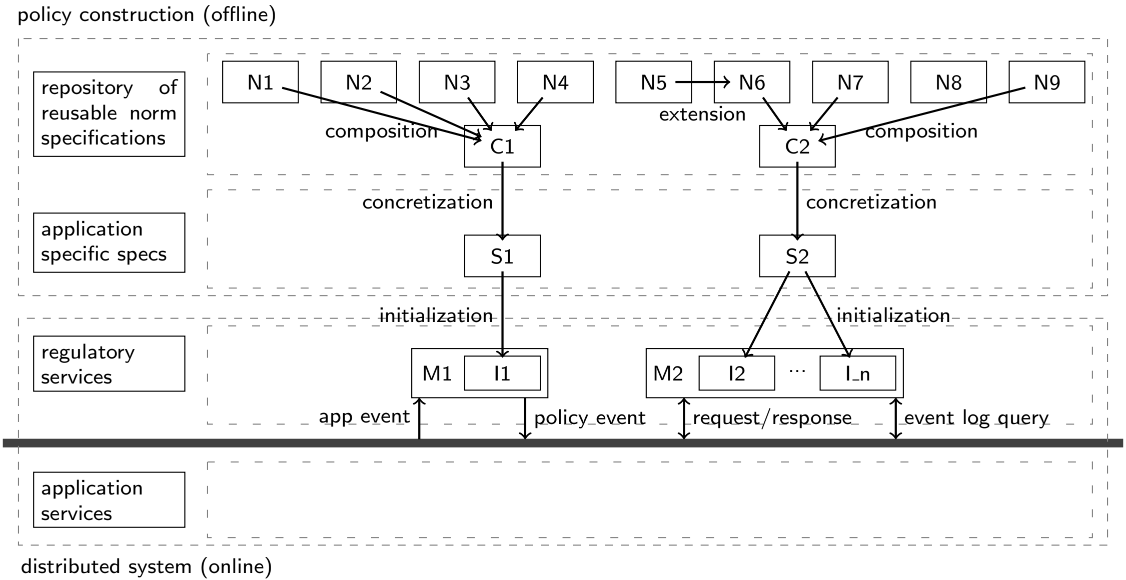
Norm-aware, distributed software systems

Regulated data exchange*:*
Data exchange systems governed by regulations, contracts and policies

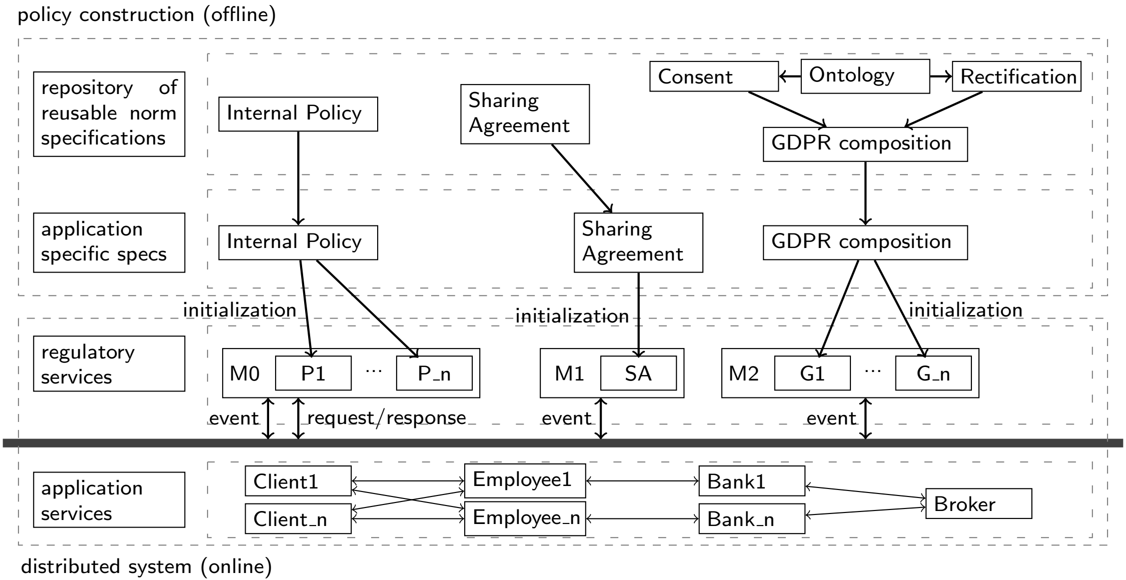
as an instance of

Regulated systems*:*
Distributed software systems with embedded regulatory services derived from norm specifications that monitor and/or enforce compliance

Regulated systems architecture



Regulated systems architecture for Know Your Customer case study



Desired properties of regulatory services

Regulatory services for: control, enforcement, monitoring and diagnosis

Explicit, formal and reusable *interpretations* of norms written as normative specifications in a high-level domain-specific language

- e.g. laws, regulations, organizational policies, contracts, codes of conduct, etc.

Explicit *qualification* of observations in terms of formalized norms

Multiple normative specifications can apply simultaneously, each having its own collection of regulatory services

Regulatory services can be dynamically updated to new versions of norms

Desired properties of norm specification language (policy language)

Formalization of norms in terms of *deontic* and *potestative* positions

- Deontic positions: Permission, prohibition, obligation
- Potestative positions: Power (ability), liability, immunity

Actors are in *normative relations* with each other:

- **power-liability** relations between a *performer* and a *recipient*
- **duty-claim** relations between a *holder* and a *claimant*

Queries produce insights into normative positions and institutional facts
Conversely, institutional facts can be validated by external services

Transitions, triggered by *input events*, modify normative positions, resulting in *output events*: e.g. new obligations, violated prohibitions, etc.

Section 2

Policy construction with eFLINT

Example – ontology

```
Fact subject
Fact data
Fact subject-of Identified by subject * data

Fact controller

Fact processor
Fact purpose
Fact processes Identified by processor * data * controller * purpose
```

Elements of the GDPR ontology

```
Fact personal-data Identified by data
  Holds when (Exists subject: subject-of(subject,data))
```

Article 4(1)

---

eFLINT: a Domain-Specific Language for Executable Norm Specifications.
L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers.
Proceedings of GPCE '20. ACM.

Example – rectification(1)

(Article 16) The data subject shall have the right to obtain from the controller without undue delay the rectification of inaccurate personal data concerning him or her. [...]

```
Fact accurate-for-purpose Identified by data * purpose

Act demand-rectification
 Actor subject
 Recipient controller
 Related to purpose
 Creates rectification-duty(controller,subject,purpose)
 Holds when (Exists data, processor:
  subject-of() && !accurate-for-purpose() && processes())
```

The data subject has the right to demand rectification of inaccurate data

Example – rectification(2)

(Article 16) The data subject shall have the right to obtain from the controller without undue delay the rectification of inaccurate personal data concerning him or her. [...]

```
Duty rectification-duty
  Holder controller
  Claimant subject
  Related to purpose
  Violated when undue-rectification-delay() // open-texture term

Fact undue-rectification-delay Identified by controller * purpose * subject

Event rectification-delay
 Related to controller, purpose, subject
 Creates    undue-rectification-delay()
 Holds when rectification-duty()
```

... rectification without undue delay ...

Example – rectification(3)

(Article 16) The data subject shall have the right to obtain from the controller without undue delay the rectification of inaccurate personal data concerning him or her. [...]

```
Act rectify-personal-data
 Actor controller
 Recipient subject
 Related to purpose
 Terminates rectification-duty(), undue-rectification-delay()
 Holds when all-processors-accurate()

Fact all-processors-accurate Identified by controller * subject * purpose
 Holds when (Forall processor, data: accurate-for-purpose()
              When processes() && subject-of())
```

Rectification

Foundations of eFLINT

(Institutional) facts, actions, events and duties are **fluents**, changing over time due to the effects of actions and events

A **specification** is a sequence of type declarations inducing a transition system. Transitions in the system are triggered by input events and produce output events. A **script** is a sequence of statements describing a trace in the transition system

Normative relations and deontic/potestative positions are inferred:

- An act-type describes a power-liability relation (if it affects normative positions)
- An action is **permitted** if it is enabled (its instance & pre-conditions hold)
- A duty-type describes a duty-claim relation
- Duty-types are used to describe **obligations** and **prohibitions**

There are only implicit references to **time**, and references are always to "now". The effects of actual time (in a running system) are triggered by input events. If necessary, a clock can be modeled using the `clock` fact and `tick()` event

Example script

```
give-consent(Alice, Bank, KYC).
collect-personal-data(Bank, Alice, A1, Advertisement).// non-compliant action
collect-personal-data(Bank, Alice, A1, KYC).           // compliant action
-accurate-for-purpose(A1, KYC).                        // e.g. Alice relocates
+accurate-for-purpose(A2, KYC).
demand-rectification(Alice, Bank, KYC).                // creates duty
?rectification-duty(Bank, Alice, KYC).                 // query succeeds
stop-processing(BankProcessor, Alice, KYC).            // data deleted
rectify-personal-data(Bank, Alice, KYC).               // terminate duty
?!rectification-duty(Bank, Alice, KYC).                // query succeeds
```

Applications of eFLINT

Automatic **case assessment** and dispute resolution

- Present: web interface on top of a command-line tool for running scripts

Policy **design** through scenario exploration

- Present: assessing sets of concrete scenarios (i.e. test suite of scripts)
- Present: scenario exploration using a command-line REPL (with backtracking)
- Future: exploring sets of scenarios satisfying certain properties (model finding)
- Future: change impact analysis (diffs between sets of scenarios)

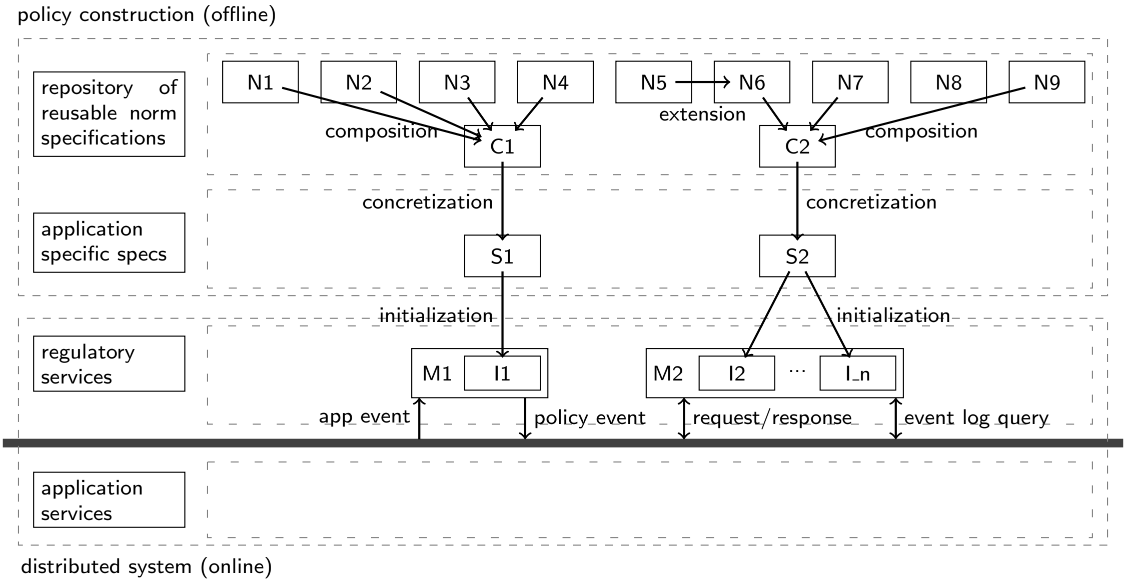
Policy **verification**

- Present: run-time checking of invariants
- In development: model checking safety and liveness properties

Online use in **regulated systems**:

- Present: TCP REPL to respond to input events and produce output events
- Present: control and enforcement using regulator actors
- In development: monitoring and diagnosis

Regulated systems architecture



Policy extension in eFLINT (offline)

Composition

eFLINT specifications are composable sets of declarations; name-conflicts are resolved:

- via encapsulation (e.g. in a module system), or
- via replacement (newer replaces older), or
- via *concretization* (more specific replaces less specific)

Concretization

A declaration C concretizes a declaration D of the same type name T when:

- C defines a subtype of D, i.e. $I_C \subseteq I_D$, or
- C is structured, D is unstructured (`data` example on next slide)

Concretizations can add derivation clauses, pre-conditions and post-conditions to a type

Example – concretization

```
Fact data
Fact subject-of Identified by subject * data
Fact purpose
```

Original declarations in GDPR ontology

```
Fact purpose Identified by KYC, Advertisement, Other

Fact client
Fact property
Fact value
Fact data Identified by client * property * value

Fact subject-of Identified by subject * data
  Derived from (Foreach data: subject-of(data.client, data))

// at most one subject is identifiable in every element of data
Invariant data-rows-not-sets :
  (Forall data, subject, subject' : subject == subject'
    When subject-of() && subject-of(subject = subject'))
```

Concretizations used in KYC case study

Section 3

Applying eFLINT in regulated systems

Online policy extension using Read Eval Print Loops (REPLs)

Sequential languages (Van Binsbergen 2020c)

In a *sequential language*, every sequence of valid programs is a valid program.
In other words, the set of programs of a sequential language forms a semi-ring

eFLINT is sequential, enabling online case analysis and policy modification

The paper has a generic *exploring interpreter* algorithm for sequential languages

Different eFLINT interfaces have been built on top of the exploring interpreter:

- A command-line interface for manual exploration
- A TCP server interface for receiving declarations and statements over a port

---

A principled approach to REPL interpreters. L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. Proceedings of Onward! '20. ACM.

From eFLINT specifications to Regulators

idea: let special 'regulator actors' execute eFLINT specifications

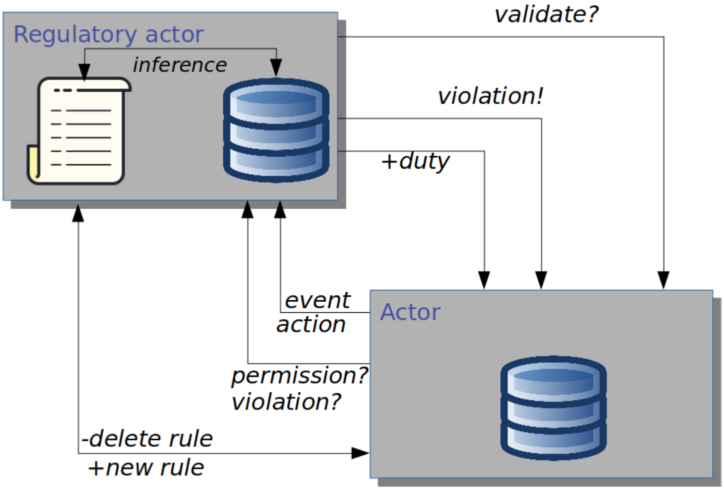
Incoming messages trigger input events

- Creating/terminating facts and triggering actions and events (statements)
  - Dynamic scenario (case) construction with automated assessment
- Creating, modifying or removing fact-, act-, event- and duty-types (declarations)
  - Dynamic policy construction
- Queries, e.g. for checking for permissions, powers and (violated) duties

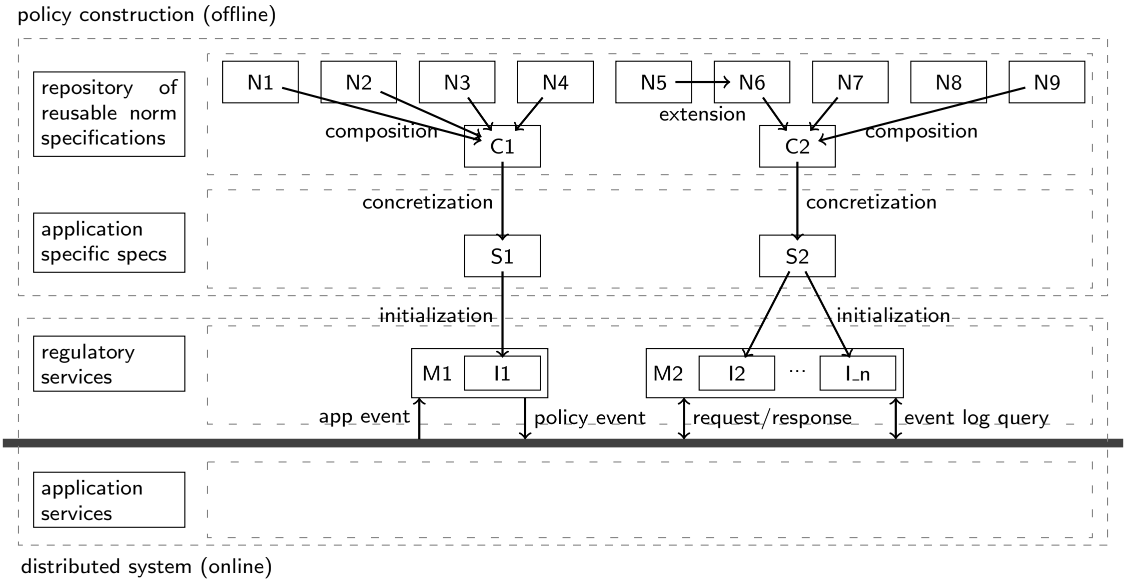
Output events trigger outgoing messages

- Notifications of new permissions and powers
- Notifications of executed (and perhaps non-compliant) actions
- Notifications of new duties and newly violated duties
- Querying an actor to determine or validate the truth of a fact

Regulator overview



Regulated systems architecture



Monitoring services

Create and maintain Regulators in response to certain application-level events:

- create Regulators by loading and initializing an eFLINT specification (e.g. contracts)
- maintain addresses of Regulators

Translate application-level events to policy-level events within correct Regulator.
Translate policy-level events from Regulators to application- or policy-level events

- Requires an intermediate or shared ontology

Request/response interactions from application- to policy-layer (and vice versa):

- A timeout value to ensure timely response
- A default response in case of timeout

Query event logs by constructing a report over past events (i.e. CloudLens DSL)

KYC – shared event ontology (GDPR compliance)

Declaration of data types, e.g. using JSON schemas to define object types

```
{
  "title"      : "ClientProfile",
  "type"       : "object",
  "required"   : [ "id", "country-code", "sbi-code" ]
  "properties": {
    "id" : {
      "type"         : "number",
      "description" : "the client for which this profile collects info"
    }
    ...
  }
}
```

Declaration of events as data types

```
APP message/timestamp:number/from:Client/to:Bank
              /{"name":"apply_for_account","KYC_consent":boolean, ...}

APP insertDB/timestamp:number/bank:Bank/contents:ClientProfile
```

KYC – monitoring GDPR compliance

```
WHEN
 message/time/client/bank
        /{"name":"apply_for_account","KYC_consent":consent,...}
NEW gdpr-contract(client, bank)
TRIGGER IN gdpr-contract(client.id,bank.id) WHEN consent == "true"
  give-consent($client.id,$bank.id,KYC). // eFLINT input event (statement)


INIT gdpr-contract(client:Client, bank:Bank) FROM "gdpr_composition.eflint"
  IDENTIFIED BY client.id, bank.id
TRIGGER
  +subject($client.id).  //eFLINT initialization statements
  +controller($bank.id).
  +processor($bank.id).


WHEN
 insertDB/time/bank/{"id":id, "country-code":country, "SBI-code":sbi, ...}
TRIGGER IN gdpr-contract(id, bank.id)
 collect-personal-data($bank.id,$id,data($id,"country",$country),KYC).
 collect-personal-data($bank.id,$id,data($id,"sbi",$sbi),KYC).
```

KYC – shared event ontology (2)

POLICY

```
illegalAction/"collect-personal-data"
             /by:Bank.id/to:Client.id/purpose:string
```

APP-REQUEST

```
permission/"collect-personal-data"
          /by:Bank.id/client:Client.id/purpose:string
```

RESPONSE

```
value:boolean/motivation:object
```

WITHIN

```
20.MILLISECONDS
```

DEFAULT

```
"false"/{"reason":"request failed"}
```

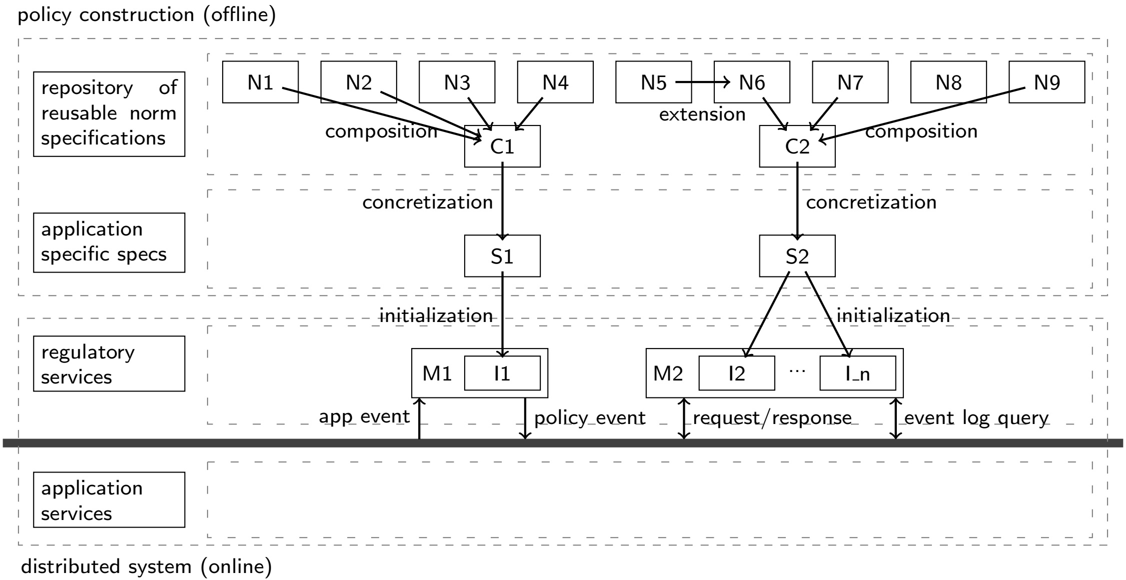
KYC – monitoring GDPR compliance (2)

```
WHEN
  ACTION-VIOLATION collect-personal-data(bank,client,purpose)
  IN gdpr-contract(client, bank)
TRIGGER
  illegalAction/"collect-personal-data"/$bank/$client/$purpose



REQUEST
  permission/"collect-personal-data"/bank/client/purpose
TRIGGER IN gdpr-contract(client.id, bank.id)
  ?Enabled(collect-personal-data($bank.id,$client.id,$purpose))
```

Regulated systems architecture



Reflections and limitations

Regulatory services can be generated from specifications

- Regulators generated from norm specifications (e.g. written in eFLINT), and
- Monitors generated from reactive interface specifications
- Verified using eFLINT TCP servers and handwritten Scala Akka code for KYC case

Reflections and limitations

Regulatory services can be generated from specifications

- Regulators generated from norm specifications (e.g. written in eFLINT), and
- Monitors generated from reactive interface specifications
- Verified using eFLINT TCP servers and handwritten Scala Akka code for KYC case

eFLINT practical and relatively easy to use for programmers, however:

- Higher-level version for domain-experts (e.g. legal experts, policy makers):
  - Language constructs for reusable, high-level patterns (design patterns)
  - Specifications directly in terms of normative positions, rather than inferred
- More restrictive version as a target for natural language processing

Reflections and limitations

Regulatory services can be generated from specifications

- Regulators generated from norm specifications (e.g. written in eFLINT), and
- Monitors generated from reactive interface specifications
- Verified using eFLINT TCP servers and handwritten Scala Akka code for KYC case

eFLINT practical and relatively easy to use for programmers, however:

- Higher-level version for domain-experts (e.g. legal experts, policy makers):
  - Language constructs for reusable, high-level patterns (design patterns)
  - Specifications directly in terms of normative positions, rather than inferred
- More restrictive version as a target for natural language processing

Limitations to presented approach for regulatory services:

- Regulators are not 'strongly reactive', handle one input event at a time
- Consequences: long computations and external validation decrease throughput
- Stateless or multi-state design as possible solutions
- Further considerations regarding the structure of policy-level events required, i.e. provide intuitive reports about current trace (explainability and diagnosis)

Section 4

Agile Software Language Engineering

Some terminology

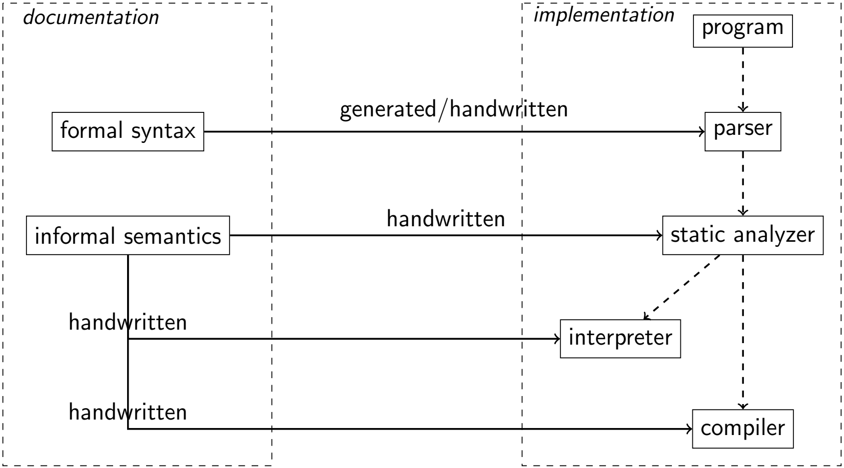
software languages: general-purpose programming languages, specification languages, modeling languages, scripting languages, domain-specific languages, meta-languages, etc...

domain-specific languages (DSLs) specialized to an application domain, ideally usable by domain experts without prior programming experience

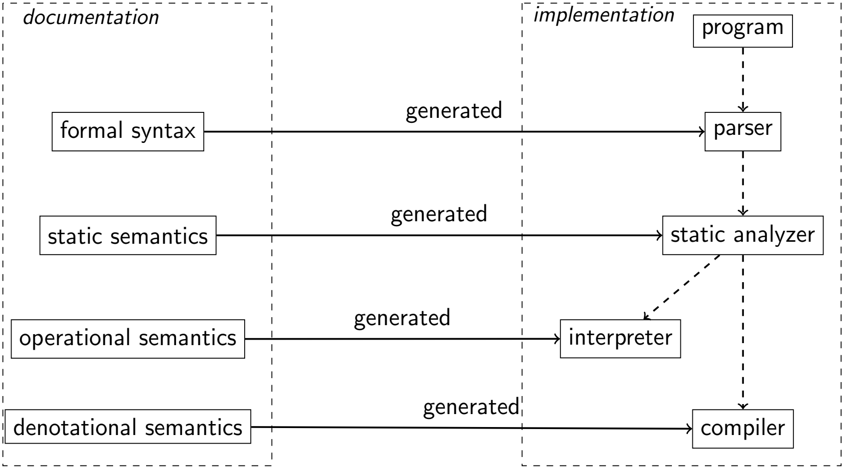
embedded DSLs (EDSLs) borrow syntax and tooling from a *host language*

meta-languages: (domain-specific) languages for constructing *object languages*

Language development – practice



Language development – ideal



Agile Software Language Engineering

To make language specifications easier to *develop*, to *maintain* and to enable *rapid prototyping*, the declarations of meta-languages should be:

modular

- A specification consists of smaller components that can be understood in isolation

compositional

- The ability to compose components and retain desirable properties

reusable

- The ability to reuse components across specifications
- Common pattern: *reuse through abstraction*
- Rapid prototyping requires *separate compilation*, i.e.
  changing one components requires only regenerating the code for that component

Contributions to generalized parsing technology

GLL Parser Combinators
(Van Binsbergen 2018,2020)

- compositional
- reuse through abstraction
- embedded

LL parsing

- modular
- separate compilation

GLL parsing
(Johnstone, Scott 2010,2013)

- compositional
- separate compilation

FUN-GLL
(Van Binsbergen, Scott 2018,2019)

- compositional
- separate compilation

CFGs
(Chomsky)

- compositional

Happy GLL back-end
(Van Binsbergen 2020)

- compositional
- reuse through abstraction
- separate compilation

LR parsing

GLR parsing
(Tomita 1985)

- compositional

Contributions to modular operational semantics

SOS
(Plotkin 1981)

M-SOS
(Mosses 2004)
- compositional

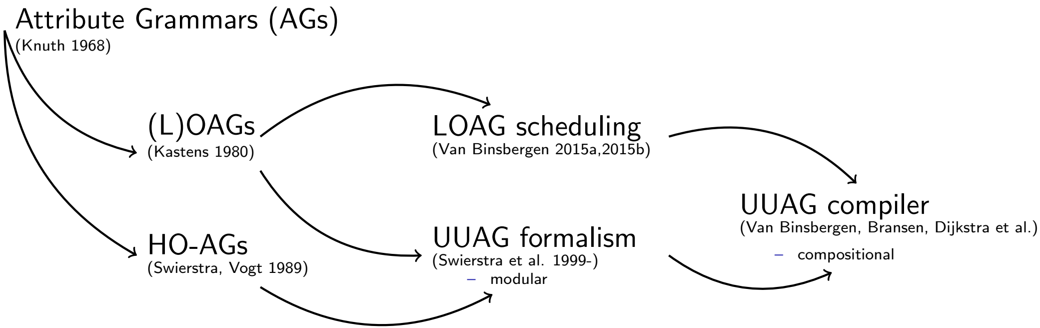
IM-SOS
(Mosses 2009)
- compositional

FunCons
(Mosses 2010-2015)
- compositional
- reuse

FunCons EDSL
(Van Binsbergen, Sculthorpe 2016,2019)
- compositional
- reuse
- embedded

Component-Based Semantics
(Van Binsbergen, Sculthorpe, Mosses 2016,2019)
- compositional
- reuse
- separate compilation

Contributions to attribute grammar scheduling



Personal toolkit of Agile Language Engineering

Generic and provably sound algorithms based on solid theory with implementations that inherit nice properties from theory

Royal Holloway, University of London & **Swansea university**:

- Executable, compositional *syntax* specification based on the FUN-GLL algorithm
- CBS meta-language for *operational semantics* with reusable FunCons
- Modular FunCon implementations generated from CBS specifications

Utrecht University:

- UUAG formalism for modular attribute grammar specifications of *static analyses*
- Pure interpreter definitions with monads or attributes for 'algebraic effects'

Centrum Wiskunde & Informatica (CWI):

- Rascal meta-language[1] for extensible *syntax*, *interpretation*,
- *denotational semantics* in terms of rewrite rules, and
- generated IDE support

[1] Developed by CWI and taught at UvA

Software languages for data exchange systems

L. Thomas van Binsbergen[1]

[1]Centrum Wiskunde & Informatica
l.t.van.binsbergen@cwi.nl

April 2020

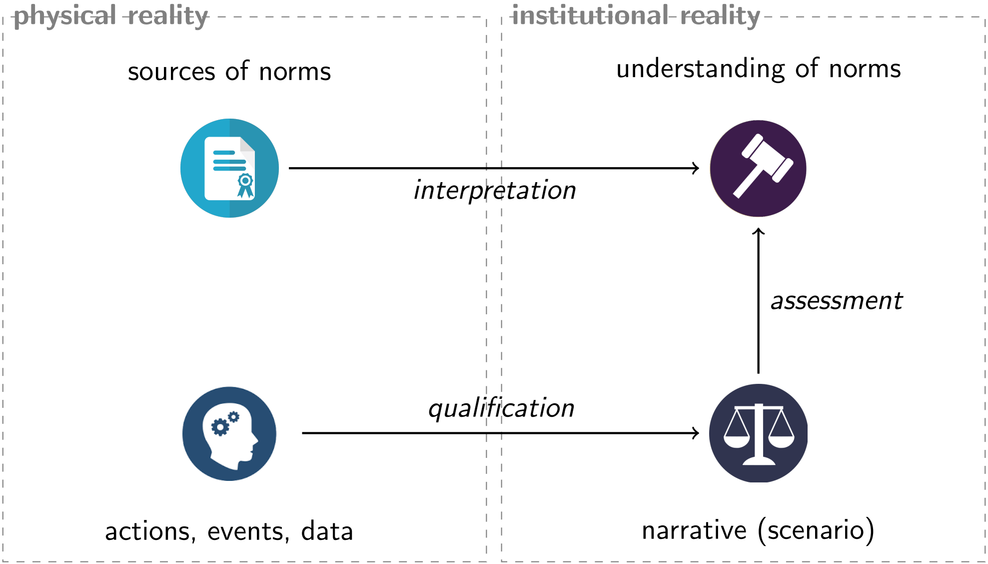
Software languages for data exchange systems

L. Thomas van Binsbergen[1]

[1]Centrum Wiskunde & Informatica
l.t.van.binsbergen@cwi.nl

April 2020

Realities


physical reality
institutional reality
sources of norms
understanding of norms
interpretation
assessment
qualification
actions, events, data
narrative (scenario)

Producing normative actors

Legal analyst / policy expert

Produces a semi-formal interpretation of relevant sources (e.g. using the FLINT language) in terms of (Hohfeldian) power-liability and duty-claim relations between actor roles, possibly aided by natural language processing and/or editorial software.

Producing normative actors

Legal analyst / policy expert

Produces a semi-formal interpretation of relevant sources (e.g. using the FLINT language) in terms of (Hohfeldian) power-liability and duty-claim relations between actor roles, possibly aided by natural language processing and/or editorial software.

Software engineer

Formalizes the semi-formal interpretation produced by the legal analyst in a high-level, domain-specific language (e.g. using the eFLINT language). The resulting interpretation can be analyzed with formal verification techniques (e.g. consistency and safety checks) and can be used to assess and compare concrete scenarios.

Producing normative actors

Legal analyst / policy expert

Produces a semi-formal interpretation of relevant sources (e.g. using the FLINT language) in terms of (Hohfeldian) power-liability and duty-claim relations between actor roles, possibly aided by natural language processing and/or editorial software.

Software engineer

Formalizes the semi-formal interpretation produced by the legal analyst in a high-level, domain-specific language (e.g. using the eFLINT language). The resulting interpretation can be analyzed with formal verification techniques (e.g. consistency and safety checks) and can be used to assess and compare concrete scenarios.

All interpretations are stored modularly, with references to sources, and under version control.

Producing normative actors

Legal analyst / policy expert

Produces a semi-formal interpretation of relevant sources (e.g. using the FLINT language) in terms of (Hohfeldian) power-liability and duty-claim relations between actor roles, possibly aided by natural language processing and/or editorial software.

Software engineer

Formalizes the semi-formal interpretation produced by the legal analyst in a high-level, domain-specific language (e.g. using the eFLINT language). The resulting interpretation can be analyzed with formal verification techniques (e.g. consistency and safety checks) and can be used to assess and compare concrete scenarios.

All interpretations are stored modularly, with references to sources, and under version control.

Application as normative actors

A specific version of a formal interpretation is concretized based on configuration options.
The concrete interpretation is compiled to the source code of a normative actor.
The normative actor is dynamic in that it can receive policy updates.

Actor-role abstraction

object-oriented programming:
Class abstractions (types) are instantiated to objects. Objects have a private state and communicate information through method calls. An object relinquishes execution control when calling a method of another object.

actor-oriented programming:
Actor-role abstractions (types) are instantiated by actors. Actors have a private state and communicate through message-passing. Actors execute concurrently, always in response to an incoming message.

Actor-role abstraction

object-oriented programming:
Class abstractions (types) are instantiated to objects. Objects have a private state and communicate information through method calls. An object relinquishes execution control when calling a method of another object.

actor-oriented programming:
Actor-role abstractions (types) are instantiated by actors. Actors have a private state and communicate through message-passing. Actors execute concurrently, always in response to an incoming message.

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala – `https://akka.io`

Actor-role abstraction

object-oriented programming:
Class abstractions (types) are instantiated to objects. Objects have a private state and communicate information through method calls. An object relinquishes execution control when calling a method of another object.

actor-oriented programming:
Actor-role abstractions (types) are instantiated by actors. Actors have a private state and communicate through message-passing. Actors execute concurrently, always in response to an incoming message.

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala – `https://akka.io`

agent-oriented programming:
Actor-oriented programming in which the actors (called agents) have mental qualities, such as beliefs, desires and intentions, and in which only certain kinds of messages are used, such as requests, offers, declines and promises