Domain-specific languages, regulated systems and sustainability

L. Thomas van Binsbergen
Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org
February 25, 2021
Section 1

Software languages and sustainability
In what ways is sustainability promoted by domain-specific languages, formal semantics and their application in regulated systems?
In what ways is sustainability promoted by domain-specific languages, formal semantics and their application in regulated systems?

Technological and social challenges:

- The continued ability to leverage software through execution, i.e. ensuring there are practical means of running a software product
In what ways is sustainability promoted by domain-specific languages, formal semantics and their application in regulated systems?

Technological and social challenges:

- The continued ability to leverage software through execution, i.e. ensuring there are practical means of running a software product
- The ability of software to adjust to changing circumstances, e.g. new execution environments (such as platforms, devices, services), new and updated regulations, and changing teams of developers/maintainers
In what ways is sustainability promoted by domain-specific languages, formal semantics and their application in regulated systems?

Technological and social challenges:

- The continued ability to leverage software through execution, i.e. ensuring there are practical means of running a software product.
- The ability of software to adjust to changing circumstances, e.g. new execution environments (such as platforms, devices, services), new and updated regulations, and changing teams of developers/maintainers.
- The continued ability to leverage the creative value put into software, i.e. can we still understand the logic of the code / the algorithm? can we extract and reuse it?
Legacy Systems

- Written in arcane, unstructured languages,
- hard to maintain and costly to migrate
- grew organically, in a non-modular fashion,
- uses non-standardised interfaces between components and other software,
- has little documentation or of poor quality,
- may require specific environments to run,
- and no one ‘owns’ the software anymore, nor understands how it does what it does

BASIC program on an old Commodore
Unlike natural languages, software languages are potentially **formal** and **exact**. However, few languages have a ‘formal contract’ between design and implementation.

Formal semantics enables such formal contracts.

Reference manuals typically have formal syntax and informal semantics.
Formalisations of general-purpose languages are complex and hard to maintain

Domain-specific languages have much smaller scopes

Figure: MySQL
Figure: PlantUML
Figure: DOT
Model-driven engineering

Generate implementations from *models* of the desired system:
- Specify the essence, abstracting away from implementation details
- Visualisation, inspection, and checking of model in isolation

*Figure:* by Johan den Haan, CTO at Mendix
Engineers typically learn individual languages by ‘speaking’ with a compiler.

Programming should be taught in terms of paradigm-agnostic concepts.
Engineers typically learn individual languages by ‘speaking’ with a compiler

Programming should be taught in terms of paradigm-agnostic concepts

The PLanCompS project: http://plancomps.org

Component-based approach towards formal, operational semantics

Main contributions of the project:

- A library of highly reusable, executable fundamental constructs (*funcons*)
- The meta-language CBS for defining component-based semantics

---

*a*https://plancomps.github.io/CBS-beta/Funcons-beta/Funcons-Index/

*b*Executable Component-Based Semantics. Van Binsbergen, Sculthorpe, Mosses. JLAMP 2019
Can this pipeline support modular, incremental DSL development?

Can funcons serve as the basis for teaching programming?
Section 2

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

as an instance of

Regulated systems:  
*Software systems with embedded regulatory services derived from norm specifications that monitor and/or enforce compliance*
Regulated data exchange:

Data exchange systems governed by regulations, agreements and policies

as an instance of

Regulated systems:

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

NWO-funded: SSPDDP – Secure and scalable, policy-driven data exchange
Regulated data exchange:

Data exchange systems governed by regulations, agreements and policies

as an instance of

Regulated systems:

Software systems with embedded regulatory services derived from norm specifications that monitor and/or enforce compliance

NWO-funded: SSPDDP – Secure and scalable, policy-driven data exchange

EFRO-funded: AMDEX Fieldlab – neutral data-exchange infrastructure
Towards regulated systems

Monolithic programs

Regulated (software) systems
Towards regulated systems

Monolithic programs \(\xrightarrow{\text{distribution}}\) Service-oriented architectures

Regulated (software) systems
Towards regulated systems

Monolithic programs \[\xrightarrow{distribution}\] Service-oriented architectures

\[\downarrow\]

AI

Autonomous systems

Regulated (software) systems
Towards regulated systems

Monolithic programs $\xrightarrow{\text{distribution}}$ Service-oriented architectures

$\downarrow \quad A I$

Autonomous systems $\xrightarrow{\text{distribution}}$ Social software systems

$\downarrow \quad A I$

Regulated (software) systems
Towards regulated systems

Monolithic programs $\xrightarrow{distribution}$ Service-oriented architectures

Autonomous systems $\xrightarrow{distribution}$ Social software systems $\xrightarrow{\text{norms/enforcement}}$ Regulated (software) systems

AI
Regulated systems – points to address

Formalization of applicable norms: reusable, modular and dynamically updateable

Different methods of embedding and enforcing norms:

- Static ex-ante: verify and apply norms during software production  
  e.g. *correct-by-construction arguments, model checking*

- Dynamic ex-ante: apply rules at run-time, guaranteeing compliance  
  permits decisions (behavioural, normative) that depend on input

- Embedded ex-post enforcement: specified responses to violations  
  permits (regulated) non-compliant behaviour, e.g. based on risk assessment by agent

- External ex-post enforcement: external responses to violations  
  e.g. *auditing, conformance checking*  
  permits (human-)intervention in running system

Production of diagnostic reports and/or audit trails to enable evaluation and reflection
Our approach to regulated systems

- **Application Services**
  - input/output
  - queries
  - monitors & notifies
  - penalizes, rewards & notifies

- **Normative Services**
  - monitors & notifies

- **Enforcement Services**
  - input/output

- **Users**
  - application services
  - regulatory services
Regulated systems – points to address

Derivation of regulatory services from formalization of norms

Interfacing between application and regulatory services:
- Monitoring (communicated and silent) behaviour of services
  *difficulties: fallible and subject to manipulation*
- Regulatory services responding to queries about normative positions
  *e.g. do I have permission to...? or the obligation to...?*
- Application services verifying facts on behalf of regulatory services
  *e.g. verifying credentials*
- Regulatory services communicating changes in normative positions
  *e.g. gaining/losing powers, holding/satisfying obligations, violations*

Challenges: different interpretations of norms and different qualifications of situations
Our approach to model-driven experimentation

Executable actor-model

System specification

- norms
- agents
- coordination
- scenarios

Connect models

Event log

Log

Compare

Generation
Our approach to model-driven experimentation

eFLINT – formalization of norms from a variety of sources
declarative reasoning about facts, actions and duties
reactive component for integration in software systems
including actor-based implementation

AgentScriptCC – specification of services as agents
reactive BDI agents,
compiled to actor-based implementation

Actor-oriented programming in the Akka framework:
https://akka.io/
actor systems modelling social software systems
eFLINT actors

- eFLINT actor
- changes in norms
- query (e.g. permission?)
- notification (e.g. violation / new duty)
- notification (e.g. of action)
- query (e.g. verification)

Actor
Our approach to model-driven experimentation

- Executable actor-model
  - A1
  - A2
  - A3
  - A4
  - A5

- System specification
  - norms
  - agents
  - coordination
  - scenarios

- Connect models

- Event log
- Logging
- Compare
- Generation
Agents are translated into actor-based micro-systems

Consisting of:
- Interface actor
- Intention pool actor
- \( n \geq 1 \) Intention actors
- Belief base actor
- Belief base
- Plan library
Case study around the Know Your Customer principle adopted by financial institutions to meet international regulations by assessing client profiles to compute risk

Involves three types of “normative documents”:

1. Sharing agreement – a contract between banks of a consortium
2. Internal policy – a sort of contract between bank and employee
3. GDPR – a sort of contract between bank and client

For each document we can describe its norms, the behaviour of relevant actors (clients, banks, employees and broker) and how the norms are enforced
(Article 1) A member of the consortium has the right to request a risk assessment computation from the broker for any (potential) client.

(Article 2) The data broker has the power to oblige members of the consortium to share information about any client the member does business with.
(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. [...]

<table>
<thead>
<tr>
<th>Act</th>
<th>demand-rectification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>subject</td>
</tr>
<tr>
<td>Recipient</td>
<td>controller</td>
</tr>
<tr>
<td>Related to</td>
<td>purpose</td>
</tr>
<tr>
<td>Creates</td>
<td>rectification-duty()</td>
</tr>
<tr>
<td>Holds when</td>
<td>(Exists data, processor: subject-of() &amp;&amp; processes() &amp;&amp; !accurate-for-purpose())</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Duty</th>
<th>rectification-duty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holder</td>
<td>controller</td>
</tr>
<tr>
<td>Claimant</td>
<td>subject</td>
</tr>
<tr>
<td>Related to</td>
<td>purpose</td>
</tr>
<tr>
<td>Violated when</td>
<td>undue-rectification-delay()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fact</th>
<th>undue-rectification-delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified by</td>
<td>controller * purpose * subject</td>
</tr>
</tbody>
</table>
From eFLINT specifications to eFLINT actors

**idea:** let ‘eFLINT actors’ administer 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. to check whether actions are permitted or duties are violated

### Output events trigger outgoing messages
- Notifications of newly permitted actions
- Notifications of executed actions and whether they were permitted
- Notifications of new duties and violations of duties
- Querying an actor to determine or verify the truth of a fact
eFLINT actors

- **Actor**
  - Changes in norms
  - Query (e.g. permission?)
  - Notification (e.g. violation/new duty)

- **eFLINT actor**
  - Inference
  - Query (e.g. verification)
  - Notification (e.g. of action)
eFLINT integration and reuse – overview

Consent → Ontology → Rectification

Composition → Reusable specification

Specialization → Specialized specification

Initialization

M \[ l_1, \ldots, l_n \]
Reusable GDPR concepts

| Fact controller  |
| Fact subject     |
| Fact data        |
| Fact subject-of  |
| Identified by    |
| subject * data   |

Specialization to application

| Fact bank          |
| Fact client        |
| Fact controller    |
| Derived from       |
| bank               |
| Fact subject       |
| Derived from       |
| client             |
| Fact data          |
| Identified by      |
| Int                |
| Event data-change  |
| Terminates data    |
| Creates data       |
| (data + 1)         |
| Fact subject-of    |
| Derived from       |
| subject-of         |
| (client, processed) |
| ,subject-of        |
| (client, data)     |
| Fact processed     |
| ...                |

Instantiation at run-time

+bank(GNB).
+client(Alice).
+data(0).

Derived after instantiation

+controller(GNB).
+subject(Alice).
+subject-of(Alice,0).
eFLINT integration – overview

Consent → Ontology → Rectification

Composition

Reusable specification

Specialization

Specialized specification

Initialization

M \[ I_1 \ldots I_n \]
WHEN
Message(client:ClientRef, bank:BankRef, req:BankTypes.ApplicationRequest)
TRIGGER
INIT gdpr(bank, client) // instantiates GDPR actor

INIT gdpr // defines constructor
WITH bank:BankRef, client:ClientRef // Scala class parameters
IDENTIFIED BY (bank.path.name, client.path.name) // pair of values as id
FROM "gdpr_specialization.eflint" // eFLINT file to load
TRIGGER // eFLINT initialization
+client(${client.path.name}). // statements
+bank(${bank.path.name}).
+data(0).

WHEN
Message(client:ClientRef, bank:BankRef, msg:BankTypes.CountryUpdate)
TRIGGER IN gdpr(bank.path.name, client.path.name)
demand-rectification(purpose=KYC). // qualified as demand
Main component: ‘plan rules’ \( E : C \Rightarrow A \)
- when event \( E \) happens
- and if condition \( C \) holds,
- then do action \( A \)

Example from **client**:
- \( E \): Agent receives the message \( \text{give\_info} \)
- \( C \): \( B \) is a bank to which client is applying or has successfully applied, \( s \) is SBI-code of client, \( c \) is country where client is based and message sender is employee of bank \( B \).
- \( A \): send SBI-code and country to original sender of \( \text{give\_info} \) message

\[
+!\text{give\_info}(B) : \text{my\_sbi}(S) \& \& \\
\text{my\_country}(C) \& \& \\
\text{employee\_of}(#\text{executionContext}.\text{sender}\.name, B) \& \& \\
(\text{applying\_to}(B) \| \text{client\_of}(B)) \Rightarrow \\
\#\text{achieve}(#\text{executionContext}.\text{sender}\.ref, \text{info}(S,C)).
\]
Our approach to model-driven experimentation

- **System specification**
  - norms
  - agents
  - scenarios
  - coordination

- **Executable actor-model**
  - A1
  - A2
  - A3
  - A4
  - A5

- **Event log**
  - logging

- **Connect models**
  - compare

- **Generation**

L. Thomas van Binsbergen

Domain-specific languages, regulated systems and sustainability
(Rule 1) An employee has the duty to perform a risk analysis on the profile of a client within four weeks of the creation or modification of the profile

---

**Employee**

```plaintext
+!interview(Client) :
  bank(B) &&
  B == #.executionContext.sender.name =>
    #achieve(Client,give_info(B)).

+!info(SBI,Country) :
  bank(B) =>
    Client = #executionContext.sender.name;
    Info = info(SBI,Country);
    +information(Client,Info);
    #achieve(B,interview_complete(Client,Info)).

+!do_risk_analysis(C,info(SBI,Country)) =>
  B = #executionContext.sender.name;
  R = #kyc.algorithms.risk(B,SBI,Country);
  #achieve(B,assign_risk(C,R)).
```

**Bank**

```plaintext
+!interview_complete(Client,Info):
  E = #executionContext.sender.name &&
  employee(E) &&
  not client(Client) =>
    #println("interview done for " + Client);
    +information(Client,Info);
    +client(Client);
    #achieve(E,do_risk_analysis(Client,Info)).
```

L. Thomas van Binsbergen  Domain-specific languages, regulated systems and sustainability  32 / 38
Example scenario execution

1. Initialization
   - hire(Risks2)
   - hire(Risks1)

2. Client ClientAgent1 Registration to BankAgent2
   - application_request(KYC)
   - interview_client(ClientAgent1)
   - information_request()
   - interview_information(Sanchez, Chili, Farming, SanchezEmail)
   - interview_complete(ClientAgent1, ClientProfile(Sanchez))
   - application_response(true)
   - assessment_duty(BankAgent2, EmployeeAgent2, ClientAgent1)
   - perform_risk_assessment(ClientAgent1)
   - risk_assessment_completed(ClientProfile(Sanchez), LOW)

3. Sharing
   - duty_to_share(ClientAgent1, Country)
   - duty_to_share(ClientAgent1, SBI)
   - to_share(ClientAgent1, SBI, Farming)
   - to_share(ClientAgent1, Country, Chili)
   - compute_risk(ClientAgent1, BankAgent1_secret_algorithm)
   - duty_to_share(ClientAgent1, Country)
   - duty_to_share(ClientAgent1, SBI)
   - to_share(ClientAgent1, SBI, Farming)
   - to_share(ClientAgent1, Country, Chili)
   - compute_risk(ClientAgent1, LOW)

4. Computation
   - do_compute(BankAgent1, ClientAgent1, BankAgent1_secret_algorithm)
   - computed_risk(ClientAgent1, LOW)

L. Thomas van Binsbergen
Conclusions

- We can produce executable models of regulated systems, by combining
  - normative actors derived from normative specifications (in eFLINT),
  - actor implementations derived from agent scripts (in AgentScriptCC),
  - queries sent to normative actors for dynamic ex-ante enforcement, and
  - enforcement actors for dynamic ex-post enforcement
- enabling experiments with norms, enforcement mechanisms and system set-ups.
Future work

**Ongoing**

- DSL development and analysis for behaviour, norm and scenario specification
- Complete generation of executable-actor models from high-level specification
- Bring modelling to practice;
  - apply models by deriving (parts of) containerized applications for use cases in our projects on data exchange: SSPDDP, DL4LD, EPI, and AMDEX
  - explainable decision making in projects with governmental organizations

**Future**

- Static analysis of (combined) models, e.g. model checking norm specification, and consistency checking between between behaviour, normative actors and scenarios
- Additional execution platforms:
  - Containerized applications, e.g. Docker and Kubernetes
  - High-performance cloud (HPC)
  - Blockchain
The complex-cyber infrastructure group of the University of Amsterdam is experimenting with regulated systems – in which norms from a variety of sources are enforced – by deriving executable models from high-level specifications.

Such systems require several kinds of enforcement mechanisms for norms, based on whether compliance can/should be/is checked before or after a violation occurs and before or after an application runs.
Reflections on sustainability

The continued ability to leverage software through its execution, i.e. ensuring there are practical means of running a software product:

- Model-driven engineering simplifies adopting new execution platforms

Standardisation and service-oriented architectures increase flexibility.

Regulatory services derived from independent, explicit formalisations of norms make it possible to adjust to changes in regulations.

The continued ability to leverage the creative value put into software, i.e. can we still understand the logic of the code / the algorithm? can we extract and reuse it?

Formal languages are technology-independent (maths/funcons as a lingua franca)

Language design based on sound principles, fundamental programming concepts and insights from human-computer interaction.
The ability of software to adjust to changing circumstances, e.g. new execution environments (such as platforms, devices, services), new and updated regulations, and changing teams of developers/maintainers

- Standardisation and service-oriented architectures increase flexibility
- Regulatory services derived from independent, explicit formalisations of norms make it possible to adjust to changes in regulations
The continued ability to leverage the creative value put into software, i.e. can we still understand the logic of the code / the algorithm? can we extract and reuse it?

- Formal languages are technology-independent (maths/funcons as a lingua franca)
- Language design based on sound principles, fundamental programming concepts and insights from human-computer interaction
Reflections on sustainability

The continued ability to leverage software through its execution, i.e. ensuring there are practical means of running a software product:

- Model-driven engineering simplifies adopting new execution platforms

The ability of software to adjust to changing circumstances, e.g. new execution environments (such as platforms, devices, services), new and updated regulations, and changing teams of developers/maintainers

- Standardisation and service-oriented architectures increase flexibility
- Regulatory services derived from independent, explicit formalisations of norms make it possible to adjust to changes in regulations

The continued ability to leverage the creative value put into software, i.e. can we still understand the logic of the code / the algorithm? can we extract and reuse it?

- Formal languages are technology-independent (maths/funcons as a lingua franca)
- Language design based on sound principles, fundamental programming concepts and insights from human-computer interaction
Domain-specific languages, regulated systems and sustainability

L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org

February 25, 2021