A language-independent methodology for compiling declarations into open platform frameworks

Paul van der Walt
http://phoenix.inria.fr

Advisor: Charles Consel

14th of December, 2015
1. Introduction

2. Methodology

3. Formalisation

4. Implementation

5. Conclusions
Problem statement

- Mobile devices extremely widespread
- ... containing ever more personal data
- Untrusted applications have access
Some perspective

An improvement!

“Google Maps” Would Like to Use Your Current Location

Don’t Allow  OK
What does this application do?
And these ones?

[Wei et al., 2012]
Remark: why focus on privacy?

- Methodology is not limited to privacy preservation
- Previously shown to work for QoS, simulation, etc. [Gatti, 2014, Bruneau and Consel, 2013]
- Privacy is a relatable motivation, highlighting consequences of design decisions
Running example: EvilCam!

Running example application.

Supposedly:

- Takes a picture
- Applies sepia filter
- Displays it to user

---

Trust me, it won't do anything evil ;)

EvilCam!™
Running example: EvilCam!

Running example application.

Supposedly:
- Takes a picture
- Applies sepia filter
- Displays it to user
- ...and shows an advert

Trust me, it won't do anything evil ;)

EvilCam!™
Running example: EvilCam!

Running example application.

Supposedly:

- Takes a picture
  → camera permission
- Applies sepia filter
- Displays it to user
- ... and shows an advert

Trust me, it won't do anything evil ;)

EvilCam!™
Running example: EvilCam!

Running example application.

Supposedly:

- Takes a picture
  → camera permission
- Applies sepia filter
- Displays it to user
- ... and shows an advert
  → network permission
Potential data flow

What one hopes:

- camera $\rightarrow$ screen
- internet $\rightarrow$ fetch advert
- nothing more.

[Do et al., 2015, Stevens et al., 2012, Felt et al., 2012]
Potential data flow

What one hopes:

- camera → screen
- internet → fetch advert
- nothing more.

Reality:

- image → stalker.net and nsa.gov
  [Do et al., 2015, Stevens et al., 2012, Felt et al., 2012]
Challenges

Guarantees:

- **Transparency**, empowering the end-user
- **Containment** of data flow
- **Conformance** of behaviour to specification

Guidance:

- **Support** for the developer with framework

[Balland and Consel, 2010]
Related Work

Static program analysis [Liu and Milanova, 2008, Elish et al., 2013, Xiao et al., 2012]
  ▶ Prefer to avoid inspecting source code (invasive, copyright)
  ▶ Frequently inaccurate, difficult problem [Rountev et al., 2004]
  ▶ Limited user transparency
Related Work

Real-time (remote) taint analysis [Enck et al., 2014]

- Not desirable on mobile devices (limited computational power)
- Lack of developer support
- Privacy concerns!
- Will not scale
Related Work

Operating system security (capability-based systems)
[Watson et al., 2010, Shapiro et al., 1999, Shapiro et al., 2004]

- Data-flow capabilities only enforced at run-time
- Major changes to existing infrastructure
- Potentially not fine-grained enough (per-app, e.g., Android)
Related Work

Language-level restrictions

ELib, W7 [Rees, 1995, Miller, 2006]
- Powerful approach, permissions per component baked into language
- Again, low adoption,
- major changes required

DiaSuite [Cassou et al., 2012], created in research team
- Specify app → generate framework
- Minimal infrastructure modification
- Previously mainly for assisted living / home automation
- Only in the context of Java!
Improving on DiaSuite

- Work builds upon DiaSuite methodology
  - No infrastructure changes required
  - Promising tailored framework approach
- Rethink the approach, without assumptions
- Delineate then explore the design space
Major thesis contributions

- Formalisation of key phases of existing DiaSuite methodology
  - To reveal design choices
  - ... and design decisions influence behaviour (example is privacy: consequences)
  - Identify key concepts. How do they map into PL concepts?
- Generalisation to language-independent methodology
  - Explore spectrum of programming languages
- Application to mobile computing domain
- Prototype implementations
1. Introduction

2. Methodology

3. Formalisation

4. Implementation

5. Conclusions
The problem with current declaration approaches

The Android model: permissions

Even with conservative permissions, behaviour is unpredictable.
The problem with current declaration approaches

The Android model: permissions

Even with conservative permissions, behaviour is unpredictable.

The DiaSuite approach: decomposition + permissions

(SCC) [Taylor et al., 2009]
How (existing) DiaSuite methodology works

[Cassou et al., 2011]
How (existing) DiaSuite methodology works

[Cassou et al., 2011]
How (existing) DiaSuite methodology works

[Cassou et al., 2011]

[Diagram]

1. Application architect specifies DiaSpec.
2. DiaSpec Compiler produces DiaSpec specification.
3. Application developer implements class...
5. End user executes Application.

DiaSpec specification

Application architect

Application developer

End user

Host Language Compiler

Tailored programming framework

Application

DiaSpec Compiler

class .... {
..................
..................
..................
}

Class .... {
..................
..................
..................
}

type system
denotational semantics

[Cassou et al., 2011]
1. Introduction

2. Methodology

3. Formalisation

4. Implementation

5. Conclusions
Example of types

Example:

1. \((\text{source Camera as Pic})\)
2. \((\text{context Filter as Pic})\)
3. \([\text{when provided Camera (get nothing)}]\)
4. \(\text{always-publish}]\)
Example of types

Example:

1. (source Camera as Pic)
2. (context Filter as Pic
   when provided Camera
   (get nothing)
   always-publish])

should result in:

1. Camera :: Pic
2. Filter :: Pic -> () -> Pic
Example of types

Example:

1. \((\text{source \ Camera \ as \ Pic})\)
2. \((\text{context \ Filter \ as \ Pic})\)
3. \([\text{when \ provided \ Camera} \ (\text{get \ nothing}) \ \text{always-publ}i\text{sh}])\)

should result in:

1. \(\text{Camera :: Pic}\)
2. \(\text{Filter :: Pic -> () -> Pic}\)

Done using PLT Redex [Felleisen et al., 2009]
Example of types

Example:

1. \((\text{source Camera as Pic})\)
2. \((\text{context Filter as Pic})\)
3. \([\text{when provided Camera (get nothing) always-publish}]\)

should result in:

1. Camera :: Pic
2. Filter :: Pic -> () -> Pic
Example of types

Example:

1. \((\text{source Camera as Pic})\)
2. \((\text{context Filter as Pic})\)
3. \([\text{when provided Camera}
   \quad \text{(get nothing)\n   \quad \text{always-publish]}\])\)

should result in:

1. Camera :: Pic
2. Filter :: Pic -> () -> Pic

Done using PLT Redex [Felleisen et al., 2009]
Example of types

Example:

1. \((\text{source Camera as Pic})\)
2. \((\text{context Filter as Pic})\)
3. \([\text{when provided Camera}]
4. \((\text{get nothing})\)
5. \((\text{always-published}))\)

should result in:

1. \(\text{Camera :: Pic} \rightarrow \text{Pic}\)
2. \(\text{Filter :: Pic} ightarrow () ightarrow \text{Pic}\)

Done using PLT Redex [Felleisen et al., 2009]
Example of types

Example:

1. (source Camera as Pic)
2. (context Filter as Pic

[when provided Camera
4. (get nothing)
5. always-publish])

should result in:

1. Camera :: Pic
2. Filter :: Pic -> () -> Pic

Done using PLT Redex [Felleisen et al., 2009]
DiaSpec recap, types

specification ::= (declaration ...)
declaration ::= (source X as τ)
 | (action X as τ)
 | (context X as τ ctxt-interact)
 | (controller X ctrl-interact)
τ ::= Bool
 | Int
 | String
 | Picture
ctxt-interact ::= [when provided Y getresource pub]
 | [when required getresource]
crtl-interact ::= [when provided Y do Z]
getresource ::= (get nothing)
 | (get Z)
pub ::= always-publish
 | maybe-publish
X, Y, Z ::= variable-not-otherwise-mentioned
DiaSpec recap, types

specification ::= (declaration ...)

declaration ::= (source     X as τ)
  | (action     X as τ)
  | (context    X as τ ctxt-interact)
  | (controller X ctrl-interact)

τ ::= Bool
  | Int
  | String
  | Picture

ctxt-interact ::= [when provided Y getresource pub]
  | [when required  getresource]

ctrl-interact ::= [when provided Y do Z]

getresource ::= (get nothing)
  | (get Z)

pub ::= always-publish
  | maybe-publish

X, Y, Z ::= variable-not-otherwise-mentioned
Type system

- Type system encodes constraints of SCC
Type system

- Type system encodes constraints of SCC

\[
\text{unique?}[X, \Gamma] \quad \vdash [\Gamma, (\text{source } X \text{ as } \tau), (\text{SRC } \tau)]
\]
Type system

- Type system encodes constraints of SCC

\[
\text{unique?}[X, \Gamma] \quad [\text{intro-src}]
\]

\[
\vdash [\Gamma, (\text{source } X \text{ as } \tau), (\text{SRC } \tau)]
\]

\[
(\text{SRC } \tau_2) = \text{lookup}[\Gamma, X_2] \quad [\text{ctx-onSrc-get-ø}]
\]

\[
\text{unique?}[X_1, \Gamma] \quad [\text{ctx-onCtx-get-ctx}]
\]

\[
\vdash [\Gamma, (\text{context } X_1 \text{ as } \tau_1 [\text{when provided } X_2 (\text{get nothing}) _]), (\text{CTX-prov } \tau_1)]
\]
Type system

- Type system encodes constraints of SCC

\[\text{unique?}[X, \Gamma] \vdash \Gamma, (\text{source } X \text{ as } \tau), (\text{SRC } \tau)]\]

\[\text{unique?}[X, \Gamma] \vdash \Gamma, (\text{context } X_1 \text{ as } \tau_1 [\text{when provided } X_2 \text{ (get nothing) } \_], (\text{CTX-prov } \tau_1))]\]

\[\text{unique?}[X, \Gamma] \vdash \Gamma, (\text{context } X_1 \text{ as } \tau_1 [\text{when provided } X_2 \text{ (get } X_3 \text{) } \_], (\text{CTX-prov } \tau_1))]\]
A note on stages

Practical question: *when* do we implement checks?
A note on stages

Practical question: *when* do we implement checks?

<table>
<thead>
<tr>
<th>Publication and types</th>
<th>Resource access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile-time</td>
<td></td>
</tr>
<tr>
<td>static fn types</td>
<td>no invalid-access crash</td>
</tr>
<tr>
<td></td>
<td>(no examples!)</td>
</tr>
<tr>
<td>Run-time</td>
<td></td>
</tr>
<tr>
<td>contracts, guards</td>
<td>more accuracy: e.g., address book entries (Android, iOS, ...)</td>
</tr>
<tr>
<td>both feasible</td>
<td>depends!</td>
</tr>
</tbody>
</table>

Note: choice need not be global.

Especially resource access is an important decision. See Ch. 7.2.
Semantics

- **Requirement**: Decouple approach from Java implementation
- **Requirement**: Clarify where choice can be made for static/dynamic checks
- Translation from DiaSpec $\rightarrow$ simply-typed lambda calculus
- Using STLC, encode the shape of the framework (intermediate language for compiler back-end)
Semantics

\[
\left[ (\text{source } X \text{ as } \tau) \right]_{eval} \leadsto \ (\lambda(\{}\{}? \ ) : \left[ \tau \right]_{type}
\]
Semantics

\[\begin{align*}
\llbracket(source \ X \ as \ \tau)\rrbracket_{eval} & \rightsquigarrow (\lambda() \ \{ \}?) :: \llbracket\tau\rrbracket_{type} \\
\llbracket(context \ X \ as \ \tau \ [\text{when provided} \ X_2 \ get \ pub])\rrbracket_{eval} & \rightsquigarrow \\
& (\lambda(x_2 :: [X_2]_{type}, \ x_3 :: [get]_get) \ \{ \}?) :: \llbracket\text{pub, } \tau\rrbracket_{pub}
\end{align*}\]
Semantics

\[
\left[\text{source } X \text{ as } \tau\right]_{\text{eval}} \rightsquigarrow \left(\lambda() \{\} ?\right) :: \left[\tau\right]_{\text{type}}
\]

\[
\left[\text{context } X \text{ as } \tau \text{ [when provided } X_2 \text{ get } \text{pub}\}\right]_{\text{eval}} \\
\rightsquigarrow \\
\left(\lambda(x_2 :: \left[\!X_2\!\right]_{\text{type}}, x_3 :: \left[\!\text{get}\!\right]_{\text{get}}) \{\} ?\right) :: \left[\text{pub, } \tau\right]_{\text{pub}}
\]

\[
\left[\text{get nothing}\right]_{\text{get}} \rightsquigarrow \text{NULL}
\]

\[
\left[\text{get } Y\right]_{\text{get}} \rightsquigarrow (\text{NULL } \rightarrow \left[\!Y\!\right]_{\text{type}})
\]
Example for Camera and Filter

\[ \texttt{[(source Camera as Pic)]}_{eval} \leadsto (\lambda() \{ \})? ) :: [Pic]_{type} \]
Example for Camera and Filter

\[
[(\text{source Camera as Pic})]_{\text{eval}} \leadsto (\lambda() \{ \}? ) :: [\text{Pic}]_{\text{type}}
\]

\[
[(\text{context Filter as Pic})
\begin{array}{l}
\text{[when provided Camera}\\
\text{(get nothing) always-publish]}\end{array}]_{\text{eval}}
\]

\[
\leadsto
(\lambda(x_2 :: [\text{Pic}]_{\text{type}}, x_3 :: ()) \{ \}? ) :: [\text{Pic}]_{\text{type}}
\]

Note: important choice here regarding static/dynamic enforcing!
<table>
<thead>
<tr>
<th></th>
<th>Introduction</th>
<th>Methodology</th>
<th>Formalisation</th>
<th>Implementation</th>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Methodology</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Formalisation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation

- We want to explore the spectrum of programming paradigms
- Investigate checks at different stages (compile-time, run-time, ...)
- Statically typed, dynamically typed
We want to explore the spectrum of programming paradigms

Investigate checks at different stages (compile-time, run-time, ...)

Statically typed, dynamically typed

Racket is a good language-experimentation tool
  • DSL experimentation
  • contract library
  • advanced module system
  • versatile: static/dynamic typing, OO, FP, ...
Contributions in this section

- Showing that methodology generalises; discovering design possibilities
- Framework design as language generation (\texttt{#lang})
  - An aside: frameworks need not only be an OO phenomenon
Racket prototype architecture

spec.rkt

#lang s-exp "framework.rkt"
(define-context Filter ...)
...

- Specification
- Macro expansion
- Implementation
Racket prototype architecture

spec.rkt

#lang s-exp "framework.rkt"
(define-context Filter ...)
...

[spec.rkt]
(provide implement run
#%module-begin)
...

- Specification
- Macro expansion
- Implementation
Racket prototype architecture

spec.rkt

```
#lang s-exp "framework.rkt"
(define-context Filter ... )
...
```

expands to

[spec.rkt]

```
(provide implement run
#%module-begin)
...
```

uses language

imlem.rkt

```
#lang s-exp "spec.rkt"
(implement Filter (lambda ...))
...
```
Application specification

- Example from the point of view of the application developer
Application specification

▶ Example from the point of view of the application developer

```scheme
#lang s-exp "framework.rkt"

;; Specifications file, webcamspec.rkt
```
Application specification

Example from the point of view of the application developer

```
#lang s-exp "framework.rkt"
;; Specifications file, webcamspec.rkt

(define-source Camera Picture) ; built-in

(define-context Filter ; name
  Picture ; return type
  [when-provided Camera]); subscribed to

;; ...
```
Application implementation

The developer does the following:

1 ;; Implementation file, webcamimpl.rkt
2 #lang s-exp "webcamspec.rkt"
Application implementation

The developer does the following:

1 ;; Implementation file, webcamimpl.rkt
2 #lang s-exp "webcamspec.rkt"
3 (implement Filter
Application implementation

The developer does the following:

```
1 ;; Implementation file, webcamimpl.rkt
2 #lang s-exp "webcamspec.rkt"
3 (implement Filter
4   (lambda (pic)

```

... process the picture
The developer does the following:

```racket
1 ;; Implementation file, webcamimpl.rkt
2 #lang s-exp "webcamspec.rkt"
3 (implement Filter
4   (lambda (pic)
5     (let* ([canvas (make-bitmap pic ..)])
6       ; ... process the picture
7       canvas))
8 ;; ...
```

But what about **conformance**? Are other components in scope? Are the types correct? **When** should we actually check?
Separation into submodules

Compartmentalise with lexical scoping: \( C \) and \( D \) cannot communicate.

\begin{verbatim}
webcamimpl.rkt
#lang "webcamspec.rkt"
(implement C f)
(implement D g)
...
\end{verbatim}

\begin{verbatim}
(webcamimpl.rkt)
(module C-module
  (define C-impl f)
  (provide C-impl))
(module D-module
  (define D-impl g)
  (provide D-impl))
...
\end{verbatim}
Implementation

So, the implement transformer expands to:

1 (module webcamimpl "webcamspec.rkt"
Implementation

So, the implement transformer expands to:

```racket
(module webcamimpl "webcamspec.rkt"
(module Filter-module racket/gui
(define/contract Filter-impl (-> bitmap? bitmap?)
(lamda-term from previous step)
(provide Filter-impl))
...)```

The generated `webcamspec` language also checks that all `define`s have `implement`s and provides run
So, the implement transformer expands to:

```racket
(module webcamimpl "webcamspec.rkt"
  (module Filter-module racket/gui
    (define/contract Filter-impl (-> bitmap%? bitmap%?)
      (-> bitmap%? bitmap%?)
    )
  )
)
```

The generated webcamspec language also checks that all defines have implements and provides run...
So, the implement transformer expands to:

```
(module webcamimpl "webcamspec.rkt"
(module Filter-module racket/gui
  (define/contract Filter-impl
    (-> bitmap%? bitmap%?)
    ;; lambda-term from previous step
  )
  (provide Filter-impl))
...)
```
Implementation

So, the implement transformer expands to:

1 (module webcamimpl "webcamspec.rkt"
2   (module Filter-module racket/gui
3     (define/contract Filter-impl
4       (-> bitmap%? bitmap%?)
5         ;; lambda-term from previous step
6     )
7     (provide Filter-impl))
8   ...)

Note: Semantics and decisions
Implementation

So, the implement transformer expands to:

```racket
(module webcamimpl "webcamspec.rkt"
  (module Filter-module racket/gui
    (define/contract Filter-impl
      (-> bitmap? bitmap?)
      ;; lambda-term from previous step
    )
    (provide Filter-impl))
  ...)
```

The generated webcamspec language also checks that all defines have implements and provides run.
Implementation

So, the implement transformer expands to:

```
(module webcamimpl "webcamspec.rkt"

(module Filter-module racket/gui

  (define/contract Filter-impl
    (-> bitmap%? bitmap%?)
    ;; lambda-term from previous step

  )

  (provide Filter-impl))

...)
```

The generated webcamspec language also

- checks that all defines have implements
- and provides run
Evaluation

- **Transparency**: allow end-user to make an informed decision
  - Finer-grained specifications
Evaluation

- **Transparency**: allow end-user to make an informed decision
  - Finer-grained specifications
- **Containment**: predict where data can end up, what it will be used for
  - Framework controls data flow and separates into submodules
Evaluation

- **Transparency**: allow end-user to make an informed decision
  - Finer-grained specifications

- **Containment**: predict where data can end up, what it will be used for
  - Framework controls data flow and separates into submodules

- **Conformance**: ensure that the behaviour of the application corresponds to the specification (Ch. 4.4)
  - Developer can only provide a valid snippet of code (contract or type checking)
Evaluation

- **Transparency**: allow end-user to make an informed decision
  - Finer-grained specifications
- **Containment**: predict where data can end up, what it will be used for
  - Framework controls data flow and separates into submodules
- **Conformance**: ensure that the behaviour of the application corresponds to the specification (Ch. 4.4)
  - Developer can only provide a valid snippet of code (contract or type checking)
- **Support**: help the developer as much as possible
  - Warnings given if application does not conform
Reflection (and eval in Racket) would allow circumventing access control
Limitations: Reflection

- Reflection (and eval in Racket) would allow circumventing access control

- Example:

```racket
(eval '(begin (require net/http-client)
   (define-values (status header response)
      (http-sendrecv "www.google.com" "/" #:ssl? 'tls))
   ...))
```

- Luckily, easy to disable
Limitations: safe module import

Lack of safe module importing

- Importing common module would allow communication
- \textit{E.g.}, context $A$ and $B$ import $M$, then write to $M.var1$
- Must be solved by run-time / OS (see ELib [Miller, 2006])
Lessons learnt

- Static types are unnecessary [Cassou, 2011]
  - E.g., compile-time resource management in dynamic language is feasible
- In fact, methodology is paradigm-independent [van der Walt et al., 2015]
- Only requirement is pre-run-time stage (Ch. 7.2 §3)
  - Examples include type system, macro stage, external compiler, ...
- Choosing the right stage to implement a check is crucial (Ch. 7.2 §2)
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
</tr>
<tr>
<td>2. Methodology</td>
</tr>
<tr>
<td>3. Formalisation</td>
</tr>
<tr>
<td>4. Implementation</td>
</tr>
<tr>
<td>5. Conclusions</td>
</tr>
</tbody>
</table>
Summary

- Open platforms are in widespread use
- Concerning privacy, current approaches fall short
  - Require major infrastructure changes
  - Do not provide insight to end-user
- Methodology is applicable to wide spectrum of programming languages
- Rich specifications enable improved guarantees and guidance
  (illustrated with privacy)
- Methodology is applicable to diverse application domains
  (not only home automation w/ sensors)
Major thesis contributions

- Formalisation of key phases of existing DiaSuite methodology
  - Requirements for open platforms
  - Type system for specifications
  - Denotational semantics for specification terms
- Generalisation to wide spectrum of languages
  - Only pre-run-time stage necessary [van der Walt et al., 2015]
- Prototype implementations [van der Walt, 2015]
  - Qualitative evaluation according to Requirements
- Application to mobile computing domain
  - Addressing major, widespread privacy concern
Perspectives

- User acceptability study [Felt et al., 2012]
- Improved run-time support (borrow from capability-based systems)
- Specifications drive static analysis [Hallett and Aspinall, 2014]
- Fully formally verified implementation (Coq, Agda, ...)


Static analysis for inference of explicit information flow.

Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.
PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA.

A security kernel based on the lambda-calculus.
PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Evaluating the imprecision of static analysis.

Towards a verified, general-purpose operating system kernel.

EROS: A Fast Capability System.


