Constraining application behaviour by generating languages

Paul van der Walt
INRIA Bordeaux, France
paul.vanderwalt@inria.fr

ABSTRACT

Writing a platform for reactive applications which enforces operational constraints is difficult, and has been approached in various ways. In this experience report, we detail an approach using an embedded DSL which can be used to specify the structure and permissions of a program in a given application domain. Once the developer has specified which components an application will consist of, and which permissions each one needs, the specification itself evaluates to a new, tailored, language. The final implementation of the application is then written in this specialised environment where precisely the API calls associated with the permissions which have been granted, are made available.

Our prototype platform targets the domain of mobile computing, and is implemented using Racket. It demonstrates resource access control (*e.g.,* camera, address book, *etc.*) and tries to prevent leaking of private data. Racket is shown to be an extremely effective platform for designing new programming languages and their run-time libraries. We demonstrate that this approach allows reuse of an inter-component communication layer, is convenient for the application developer because it provides high-level building blocks to structure the application, and provides increased control to the platform owner, preventing certain classes of errors by the developer.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures, languages, patterns*

General Terms

Languages, Security

Keywords

Programming frameworks, functional programming, embedded domain-specific languages, sports equipment

1. INTRODUCTION

Among programming frameworks intended to be used by third party developers, we see a trend towards including mechanisms restricting access to certain features, or otherwise constraining behaviour of the application [4,17]. In the case of platforms like Android [16], the aim is usually to protect the user's sensitive data (*e.g.,* contact list, physical location) from undesired use, while still giving applications access to the resources, whether hardware or data, needed to function correctly. For example, an email application legitimately requires access to the Internet, but for a calculator application this should raise suspicion. In the case of Android, these restrictions are enforced via run-time checks against a permissions file called the Manifest, which the user accepts at install-time. Other frameworks are also adopting such declarations, in various forms [3,8].

1.1 Declaration-driven frameworks

Generally speaking, we identify a class of programming frameworks in widespread use, which we call *declaration-driven frameworks*. These frameworks are different to traditional static programming frameworks in that they have some form of declarations as input. Examples abound, including the Android SDK with its Manifest file, or the Facebook plugin SDK, both of which require permissions to be granted *a priori*. The declarations vary greatly in expressiveness, on a spectrum from simple resource permissions, *e.g.,* access to list of friends and the camera, to very expressive, *e.g.,* rules for the control flow of the application, a list of components to be implemented, *etc.* An example of the latter is DiaSuite [2], where the individual components of the application, as well as their subscription relations, are laid out in the declarations. Access to resources is also granted on a component level. These rich declarations encourage separation of components, and provide relative clarity for the user regarding potential information flow, when compared to a simpler list of permissions. Diagrams with potential information flow can be extracted from the specifications, and presented in graphical format, for example.

1.2 Problem

We identify a number of shortcomings with the systems mentioned above. Most frequently, the declarations are no more than a list of permissions checked at run-time, leaving the user guessing about the actual behaviour of the application [23]. The fact that these checks are dynamic also leads to the application halting on Android if a developer tries to access a forbidden resource. Existing static approaches that exist, on the other hand, generally try to solve different problems than resource access control, for example just checking that all required components are implemented [17].

These shortcomings are addressed by DiaSuite's approach allowing static checks on resource access, which is based on an external DSL. However, the current implementation of DiaSuite generates Java boilerplate code from the declarations, tailored to the specific application. With this generative approach, extending the declaration language would involve

modifying the standalone compiler, and in general, generated code tends to be difficult to debug and inconvenient to work with.

On the other hand, the language building platform provided by Racket allows simple implementation of an embedded DSL, with all the features of Racket potentially available to the application developer. Providing expressive constructs for specification and implementation of an application raises the level of abstraction, and allows us to implement static guarantees of resource access equivalent to DiaSuite. Furthermore, we need not maintain parser and compiler machinery in parallel with the framework infrastructure.

In this report we demonstrate the use of Racket's language extension system [21] allowing us to easily derive tailored programming environments from application declarations. This decreases effort for the application developer and gives more control to platform owners. To the best of our knowledge this is the first implementation of an EDSL which itself gives rise to a tailored EDSL in Racket. The code presented in this report is available from `http://people.bordeaux.inria.fr/pwalt`.

Outline. After giving a brief overview of related work in Sec. 2, we introduce the platform we have chosen as the basis of our prototype, as well as the example application to be implemented in Sec. 3. In Sec. 4 we show how a developer using our system would write the example application. Sec. 5 goes into detail about how the declarations unfold into a language extension, and finally in Sec. 6 we discuss strengths and weaknesses of the approach using Racket. Our conclusions are presented in Sec. 7.

2. RELATED WORK

The work most closely resembling ours is DiaSuite [1], since it is the inspiration for our approach. The relative advantages and disadvantages are thoroughly dealt with in Sec. 3.

Other than that, it seems there is not much literature on the generation of frameworks, although to varying degrees frameworks which depend on declarations are becoming ever more widely adopted [3,16]. These generally address demonstrated threats to user safety [6, 13, 15, 18, 22].

Many approaches have been proposed to address these leaks, such as parallel remote execution on a remote VM where a dynamic taint analysis is running [7]. This naturally incurs its own privacy concerns, as well as dependence on a connection to the VM. Another approach which bears similarity to our aim, is the work by Xiao *et al.* [23], which restrains developers of mobile applications to a limited external DSL based on TouchDevelop [14], from which they extract information flow via static analysis. This information is then presented to the user, to decide if the resource usage seems reasonable. This is a powerful and promising approach, but we believe that it is preferable to declare information flow paths *a priori* and constrain the developer, than having to do a heavy static analysis to extract that same information – especially since it means a developer cannot use a general-purpose language they are already familiar with, but must learn a DSL which is used for every aspect of the implementation.

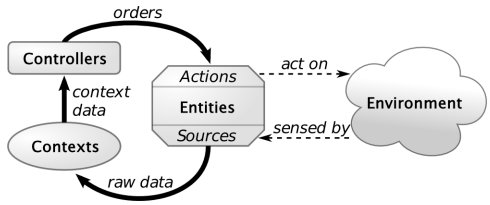

Figure 1: The *Sense/Compute/Control* paradigm. Illustration adapted from [2].

```
Declaration -> Resource | Context | Controller
Type        -> Bool | Int | String | ...
Resource -> (source srcName | action actName) as Type
Context  -> context ctxName as Type CtxtInteract
CtxtInteract -> when ( required GetData?
                     | provided (srcName | ctxName)
                        GetData? PublishSpec)
GetData       -> get (srcName | ctxName)
PublishSpec   -> (always | maybe) publish
Controller    -> controller ctrName ContrInteract
ContrInteract -> when provided ctxName do actName
```

Figure 2: Declaration grammar. Keywords are in bold, terminals in italic, and rules in normal font.

Compared to these alternatives, providing a "tower of languages"-style solution [11] seems be a good trade-off between restrictions on the developer and versatility of the implementation.

3. CASE STUDY

DiaSuite, the model for our prototype, is a declaration-driven framework which is dedicated to the *Sense/Compute/Control* architectural style described by Taylor *et al.* [19]. This pattern ideally fits applications that interact with an external environment. SCC applications are typical of domains such as building automation, robotics, avionics and automotive applications, but this model also fits mobile computing.

3.1 The Sense-Compute-Control model

As depicted in Fig. 1, this architectural pattern consists of three types of components: (1) *entities* correspond to managed[1] resources, whether hardware or virtual, supplying data; (2) *context components* process (filter, aggregate and interpret) data; (3) *controller components* use this information to control the environment by triggering actions on entities. Furthermore, all components are reactive. This decomposition of applications into processing blocks and data flow makes data reachability explicit, and isolation more natural. It is therefore well-suited to the domain of mobile computing, where users are entrusting their sensitive data to applications of dubious trustworthiness.

3.2 Declaration language

The minimal declaration language associated with DiaSuite is presented in Fig. 2. It is adapted from [2], keeping only essential constructs. An application specification is a list of

[1]Managed resources are those which are not available to arbitrary parts of the application, in contrast to basic system calls such as querying the current date.

`Declarations`. Resources (such as camera, GPS, *etc.*) are defined and implemented by the platform: they are inherent to the application domain. Context and controller declarations include interaction contracts [1], which prescribe how they interact. A context can be activated by either another component requesting its value (`when required`) or a publication of a value by another component (*i.e.,* `when provided` *`component`*). When activated, a context component may be allowed to pull data (denoted by the optional `get`). Note that contexts which may be pulled from must have a `when required` contract. Finally, a context might be required to publish when triggered (defined by `PublishSpec`). Note that `when required` contexts have no publish specification, since they are only activated by pulling, and hence return their values directly to the component which polled them. When activated, controller components can send orders, using the actuating interfaces of components they have access to (*i.e.,* `do` *`actName`*), for example printing text to the screen or sending an email.

DiaSuite compiles the declarations, written in an external DSL, into a set of Java abstract classes, one for each declared component, plus an execution environment to be used with the classes which extend them. The abstract classes contain method headers which are derived from the interaction contracts, and constrain the input and output of the developer's implementation of each component. Additionally, access to resources is passed in as arguments to these methods, so that the only way a developer may use a resource is via the capability passing method from the framework.

This approach allows advantages such as static checks by the Java compiler that the application conforms to the declarations. From these declarations it directly follows which sensitive resources components should have access to, giving the application developer a much more concise API to work with. For example, if a component only has access to the network, it need not have the API for dealing with the camera in scope. This is in contrast to Android, where all system API calls are always available, increasing the amount of information the developer must keep in mind. The disadvantage is that this is an external DSL, and thus requires a separate compiler to be maintained. It also implies less versatility, having to re-invent the wheel, and a symbolic barrier, as argued by Fowler [12].

3.3 Example application

As our running example, we use a prototype mobile application. We pretend that it is distributed for free, supported by advertisements. It allows the user to capture pictures and then view them with a colourful filter (see Fig. 3). An advertisement will be downloaded from the Internet, but we would like to prevent the developer from being able to leak the picture (which is private) to the outside world, whether intentionally or by using a malicious advertisement provider. It has been shown that this is in fact a threat: frequently, included third party advertisement libraries try to exfiltrate any private data to which they are able to get access [18].

From the specification it follows that it should be impossible for the picture to leak to the Web, since the bitmap processing component is separate from the advertisement component.

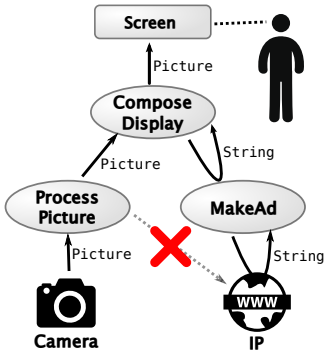

Figure 3: Simplified schematic of example application's design. We do not want the picture to be able to leak to the WWW. Note that pull requests (the curved arrows) are not parameterised, and are only used to return values.

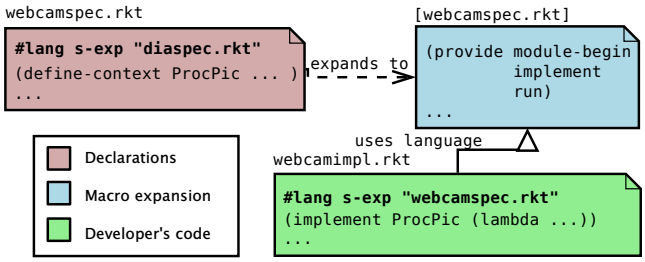

Figure 4: The prototype's architecture. Provided declarations are transformed into a tailored language for the implementation. The `implement` macro gets cases for each declared component.

4. IMPLEMENTATION OF EXAMPLE

Inspired by the DiaSuite approach, where a framework is generated from the specifications, the first step in our implementation is to provide an embedded DSL for writing specifications. It should include constructs for defining contexts and controllers, according to the grammar in Sec. 3. As illustrated in Fig. 4, when the specifications are evaluated, they in turn form a language extension which should be used to implement the application. The programming environment that is thus created provides the developer with tailored constructs for the application that is to be built, including an API precisely matched to what each component may do. In our prototype, we consider the advert developer and application developer as potentially the same, since we expect the advertisement library to provided in the form of a snippet of code that will be included along with the rest of the implementation code. This way, the advertisement code does need to be specially analysed, but is subject to the same constraints as any other code provided by the developer. That is, it can only access entities specified in the declarations.

4.1 Example specifications

The specifications as rendered in Racket, for our example application, are shown in Fig. 5. The syntax closely matches the DiaSuite declaration language previously introduced, and

```
;;; Specifications file, webcamspec.rkt
#lang s-exp "diaspec.rkt"
(define-context MakeAd String [when-required get IP])
(define-context ProcessPicture Picture
  [when-provided Camera always_publish])
(define-context ComposeDisplay Picture
  [when-provided ProcessPicture get MakeAd
     maybe_publish])
(define-controller Display
  [when-provided ComposeDisplay do Screen])
```

Figure 5: Complete declarations of the example application, in Racket prototype.

```
;;; Implementation file, webcamimpl.rkt
#lang s-exp "webcamspec.rkt"
(implement ComposeDisplay
  (lambda (pic getAdTxt publish nopublish)
    (let* ([canvas  (make-bitmap pic ..)]
           [adTxt   (getAdTxt)])
      (cond [(string=? "" adTxt) (nopublish)])
      ; ... do magic, overlay adTxt on pic
      (publish canvas))))
... ; the remaining implement-terms
```

Figure 6: The implementation of the `ComposeDisplay` context.

reflects the graphical representation of the application in Fig. 3.

4.2 Semantics of declarations

In this section, we explain the semantics of each term, from the point of view of the application developer.

The keywords `define-context` and `define-controller` are available for specifying the application, and upon evaluation, will result in a macro `implement`, for binding the implementations of components to their identifiers. For the developer this is convenient, since they only need to provide implementation terms while the framework takes care of inter-component communication as specified in the declarations. From the point of view of the framework, it provides more control over the implementation: before execution static checks can be done to determine if the terms provided by the application developer conform to the specifications.

Declaring a component C adds a case to the `implement` macro. Now, a developer can use the form `(implement` C f`)` to bind a lambda function f as the implementation of C. However, not just any f may be provided, as the arguments to `implement` are subject to a Racket function contract [5]. Unfortunately there is a name conflict between interaction contracts for components (as in DiaSuite) and function contracts in Racket, which are not the same thing. Function contracts in Racket are flexible annotations on definitions and module exports, which perform arbitrary tests at runtime on the input and output of functions. For example, a function can be annotated with a contract ensuring it maps integers to integers. If the function receives or produces a non-integer, the contract will trigger an error. The contract on f is derived from the interaction contracts of Fig. 5 as follows.

Activation conditions. These define the first argument to the function f.

`when-provided` *`x`*. First argument gets type of x. For *`ComposeDisplay`*, the contract starts with `(-> bitmap%? ...)`,[2] since it is activated by *`ProcessPicture`* publishing a bitmap image.

`when-required`. No argument added – the context was activated by pull.

[2]In reality, `bitmap%?` is shorthand for `(is-a?/c bitmap%)`, the contract builder which checks that a value is an object of type `bitmap%`.

Data sources and actions. These determine the (optional) next argument to the developer's function. This is a closure providing proxied (that is, surrounded by a run-time guard) access to the resource. This makes it convenient for a developer to query a resource, and allows the framework to enforce permissions. Actions for controllers are provided using the same mechanism.

`get` *`x`*. The contract of the closure becomes `(-> t?)` where t is the output type of x. Note that there is no parameter, just a return value. This means that a component requesting a value from another cannot exfiltrate data this way. The full contract so far is therefore `(-> ... (-> t?) ...)`.

`do` *`x`*. The contract of the closure becomes `(-> t? void?)` where t is the input type of x. The full contract is therefore `(-> ... (-> t? void?) void?)`. The final `void?` reflects that controllers do not return values.

Publication requirements. These determine the last arguments to the function contract of a context, corresponding to the output type of the context. Publishing is handled using continuations, to give us flexibility in the number of "return" statements provided.

`always_publish`. One continuation function corresponding to publication: the final contract becomes `(-> ... (-> t? void?) none/c)`, with t the expected return type.

`maybe_publish`. Two continuations to f, for publish and no-publish. The first has the contract `(-> t? void?)` with t the output type. The second continuation simply returns control to the framework. If the developer chooses not to publish, they use the second, no-publish continuation. The contract is therefore `(-> ... (-> t? void?) (-> void?) none/c)`.

The `none/c` contract accepts no values: this causes a runtime error if the developer does not use one of the provided continuations.

4.3 The implementation of the application

In Fig. 6, we show a developer's possible implementation of the context `ComposeDisplay`, which composes the modified image with the advertisement text. Essentially, a developer uses **`implement`** to bind their implementation to the identifier introduced in the specifications, *c.f.* Fig. 5. Their

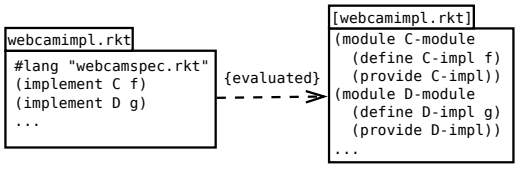

Figure 7: Separation of components using modules. The developer's code (left), and its expanded form (right). f in C cannot access D or g, because of lexical scoping.

```
1  (module webcamimpl "webcamspec.rkt"
2    (module ComposeDisplay-module racket/gui
3      (define/contract ComposeDisplay-impl
4        (-> bitmap%? (-> string?) (-> bitmap%? void?)
5            (-> void?) none/c)
6        (lambda (pic getAdTxt publish nopublish)
7          (let* ([canvas  (make-bitmap pic ..)]
8                 [adTxt   (getAdTxt)])
9            (cond [(string=? "" adTxt) (nopublish)])
10           ; .. do magic, overlay adTxt on pic
11           (publish canvas))))
12     (provide ComposeDisplay-impl))
13   ...)
```

Figure 8: The developer's code snippet is transformed into a submodule, as a result of evaluating Fig. 6. The shaded code is simply the term the developer provided in Fig. 6.

implementation should be a lambda term which obeys the contract resulting from the specification. For example, the `ComposeDisplay` context has the contract `(-> bitmap%? (-> string?) (-> bitmap%? void?) (-> void?) none/c)`. This is because it is activated by `ProcessPicture` publishing an image, it has **`get`**-access to the `MakeAd` component which returns a string, and it may optionally publish an image on account of its **`maybe_publish`** specification. The last two arguments correspond to publishing `(-> bitmap%? void?)` and not publishing `(-> void?)` continuations. The lambda function provided by the developer in Fig. 6 conforms to this contract. We see that if the advertisement component returns an empty string (line 7) the developer decides not to publish, but otherwise the string is overlaid on the picture and the developer publishes the composite image (line 9).

To prevent implementations of different components communicating outside of the condoned pathways, the `implement` macro wraps each f in its own submodule. As illustrated in Fig. 7, due to lexical scoping these do not have access to surrounding terms, but merely export the implementation for use in the top-level module. The result of this wrapping is shown in Fig. 8. The code in grey is precisely the term provided in Fig. 6, but it is now isolated from the implementations of the other components, preventing the developer from accessing them, which would constitute a leak.

Note that alongside this snippet, the rest of the implementations of the declared components must be provided in one module. This module must be implemented using the new `webcamspec.rkt` language – the one arising from the specifications we have written. The implementation module is checked before run-time to contain exactly one `(implement` C `...)` term for each declared C. However, we focus on this single context implementation to illustrate what transformations are done on the developer's code.

When the developer has provided implementations for each of the declared components, they can use the `(run)` convenience function which is also exported by the module resulting from the specifications. In the next section, we illustrate how these macros function.

5. THE FRAMEWORK AND RUN-TIME

Now that we have seen the user interface (*i.e.,* that which the application developer deals with) for our framework, we elucidate how the framework is implemented. This is broken down into a number of main parts: (1) the operation of the `define-context` and `define-controller` macros, (2) the expansion of the `implement` macro, and (3) how the run-time support libraries tie the implementations together to provide a coherent system. These mechanisms are explained globally here, though certain implementation details are elided. Notably, getting all the needed identifiers we had introduced to be available in the right transformer phases and module scopes was complicated. We invite the reader to experiment with the prototype code – the functionality for (1) and (2) is in the `diaspec.rkt` module, the run-time library can be found in the `fwexec.rkt` module.

5.1 What happens with the declarations?

Previously we saw that the first step for a developer is to declare the components of their application using the `define-context` and `define-controller` keywords. The specification should be provided in a file which starts with a **`#lang`** `s-exp "diaspec.rkt"` stanza, which causes the entire syntax tree of the specification to be passed to the function exported from `diaspec.rkt` as **`#%module-begin`**. This function does pattern matching on the specifications, and passes all occurrences of `define-` keywords to two handlers: (1) to compute and store the associated contracts, and (2) to instantiate a struct which will later store the implementation. The introduced identifiers are also stored as a list in the syntax transformer environment, *c.f.* the "Persistent effects" system presented in [20]. This compile-time storage will later be used to check implementation modules: have all components been implemented, and are all the identifiers used in the implementation declared in the specification?

To illustrate, Fig. 9 shows the expansion of the `ComposeDisplay` declaration, from line 6 of Fig. 5. Simplifications have been made, and module imports *etc.* have been omitted for brevity. Some elements which are not specific to this declaration term have been elided, namely a helper macro which transforms `(implement x ..)` terms into `(implement-x ..)`, to correspond with the generated macro in line 17, and a function which checks that all declared components have a corresponding `implement` term. Finally, we also omit the generated syntax for `module-begin-inner` from the specifications, since it is not particularly instructive. Note that it is this definition which allows the implementation module to use the specification module as its language, with the **`#lang`** `s-exp "webcamspec.rkt"` directive.

Line 1 marks the start of the implementation module, called

```
1 (module webcamspec "diaspec.rkt"
2  (define ComposeDisplay
3    (context 'ComposeDisplay
4      (interactioncontract ProcessPicture MakeAd
5        'maybePublish) 'pic))
6  (provide ComposeDisplay)
7  (module+ contracts
8    (define ComposeDisplay-contract
9        (-> bitmap%? (-> string?)
10            (-> bitmap%? void?) (-> void?) none/c))
11   (provide ComposeDisplay-contract))
12 (define-struct/contract ComposeDisplay-struct
13   ([spec   (or/c context? controller?)]
14    [implem (-> ...)])) ; contract from line 9
15 (provide ComposeDisplay-struct
16          implement-ComposeDisplay)
17 (define-syntax (implement-ComposeDisplay stx)
18  (syntax-case stx (implement-ComposeDisplay)
19   [(_ f)
20     #'(begin
21       (module ComposeDisplay-submodule racket/gui
22         (require (submod "webcamspec.rkt" contracts))
23         (provide ComposeDisplay-impl)
24         (define/contract ComposeDisplay-impl
25                          ComposeDisplay-contract f))
26       (require (submod "." ComposeDisplay-submodule))
27       (set-impl 'ComposeDisplay ; add to hashmap
28         (ComposeDisplay-struct ComposeDisplay
29            ComposeDisplay-impl)))]))
30
31 (provide run (rename-out
32                (module-begin-inner #%module-begin)))
33 (define-syntax (module-begin-inner stx2)
34    ... )) ;; omitted
```

Figure 9: The simplified expansion of the specifications, concentrating on `ComposeDisplay` from Fig 5. This code corresponds to the blue box in Fig. 4.

`webcamspec`. It still references `"diaspec.rkt"`, which is the language the specification was written in, *c.f.* Fig. 5. This leaves us with the code resulting from the `ComposeDisplay` context. In line 2, we see that a binding is introduced, using the name the developer chose for the component. Its value is a representation of the declaration, and is used to derive the contract. In line 7, a submodule is appended with the Racket contract the implementation is expected to adhere to. The **`module+`** keyword adds terms to a named submodule, creating the submodule if necessary [10]. Line 12 defines a tailored struct: it will hold the implementation of `ComposeDisplay`, in the field tagged with the corresponding contract. It becomes more interesting in line 17, where we see that the `implement` keyword wraps the developer's implementation in an independent submodule, as explained previously. This submodule will not have access to the surrounding scope, hence the need for the `contracts` submodule, which we import in line 22. As an aside, the **`#'`** form is shorthand for **`syntax`**, which is similar to **`quote`**, but produces a syntax object decorated with lexical information and source-location information that was attached to its argument at expansion time. Crucially, it also substitutes `f`, the pattern variable bound by **`syntax-case`** in line 19, with the pattern variable's match result, in this case the developer's implementation term. Next, in line 26, we have left the scope of the submodule. We **`require`** the submodule, bringing `ComposeDisplay-impl` into scope, which we add to a hash map (line 27). This hash map associates names of components to their implementations. Note how we are using the previously-defined struct, which forces the implementation term to adhere to its contract.

As a side-effect, `run` is only available to the developer if they manage to evaluate the implementation module without compile errors, which implies that only valid specifications and implementations allow the developer to execute the framework. Since the implementation of the framework run-time library is mundane, we do not discuss it here. To run this code, `racket-mode`[3] or DrRacket[4] [9] can be used. Simply load the `webcamimpl.rkt` file, and when it is loaded, evaluate `(run)` in the REPL.

6. EVALUATION AND DISCUSSION

In the end, the application developer is presented with a reasonably polished and coherent system for implementing an application in two stages, which allows the platform to give the user more insight into what mischief an application could potentially get up to. This assumes that the specifications are distributed with the application, and presented to the user (optionally formatted like Fig. 3), and that the software is compiled locally, or on a server that the user trusts. This would ensure that the implementation does indeed conform to the specifications.

We observe that going beyond Racket the functional programming language, and using it as a language-building platform, is where it really shines. We can mix, match and create languages as best fits the niche, then glue modules together via the common run-time library provided by Racket. This allows great flexibility and control, since with Racket's

[3]Tested using MELPA version 20150330.1125 of Greg Hendershott's wonderful Racket mode for Emacs.
[4]Tested using DrRacket v6.1.1.

#lang mechanism, we can precisely dictate the syntax and semantics of our new languages. These two aspects therefore give Racket a lot of potential in the emerging domain of declaration-driven frameworks.

6.1 Limitations

Unfortunately, there are issues that would need to be resolved before the proposed approach would be feasible in the real world. One of the trickiest parts of ensuring no communication between components is that consequently we cannot allow a developer to use any external modules in their code. This is because if a developer could **require** any module, they could in effect execute arbitrary code. It could also be used as a communication channel, since modules have mutable state. Therefore, in the prototype, we chose not to allow any importing of modules, but for a realistic application this would probably not be acceptable – we could imagine needing to use a library for parsing JSON, or processing images, or any number of benign tasks. Perhaps this would be a decision for the platform provider to make: is a particular library "safe" and could it be white-listed?

Another potential leak could be the **eval** form. Using it, a developer could easily obfuscate any behaviour desired. In fact, arbitrary imports and calls would be possible that way. We therefore inspect a developer's implementation for such things as the use of **eval**, and reject them syntactically, but since the binding might be hidden or renamed, this approach is not necessarily robust. This highlights a need in Racket: allowing components or functions to be pure would solve this vulnerability. Perhaps Typed Racket [20] will offer a solution in the future – purity analysis is on the project to-do list.[5]

It was also rather finicky to implement all the macros as described above. Although conceptually simple, it turns out to be pretty difficult in practice to get all the identifiers to be available in the right syntax transformer phases. The macro debugger in DrRacket is quite powerful, but unfortunately still leaves a lot to be desired. For example, we failed to get it to show the completely expanded implementation module as it is presented in Fig. 9 – that code is largely worked out by hand. Finer control over the macro unfolding would therefore be beneficial.

The end result is not at all pessimistic, in spite of these shortcomings and difficulties. The prototype does demonstrate the power of a language such as Racket, which gives a programmer the capability to easily modify syntax and provide custom interpretations. This prototype also demonstrates that it is possible to cleanly separate concerns and enforce a certain structure on the final implementation.

6.2 Future work

There are a number of clear avenues for improving this work. Firstly, we note that the chosen platform and model are merely examples, it should be easy to build similar "active" specification DSLs for other domains. This modular approach is also very flexible: we could choose to use any Racket extension as the implementation language for the developer to use, whether it be FRTime or Typed Racket or any other of the many libraries. We could even decide to provide different languages for different modules – the changes would be minor. If for example Typed Racket were to support purity analysis in the future, this would be a very attractive option, allowing us to be confident that no unwanted communication between components is possible. As stated, though, before this approach could be introduced into the wild, a safe module importing mechanism should be devised.

Another aspect to be dealt with is a very practical one: how to integrate this approach into an application store, where users could download applications for use on their local platforms. As it stands, the developer would have to submit their specification and implementation modules as source code, and the application store would need to compile them together, to ensure that the contracts and modules have not been tampered with. The application store – which the user would have to trust – could then distribute compiled versions of the application which would be compatible with the run-time library locally available on users' devices. Clearly, this is not be desirable in all situations: most commercial application developers submit compiled versions of their software, which in our case could allow them to *e.g.,* modify the contracts, rendering the applications unsafe.

[5] The page "Typed Racket Plans" at `https://github.com/plt/racket/wiki/Typed-Racket-plans` gives us hope.

7. CONCLUSION

In conclusion, we have tried to address the problem of resource access control to protect privacy of end users' sensitive data. Taking inspiration from the DiaSuite approach, we demonstrate an embedded DSL for specifying applications, which itself unfolds to a programming environment, placing restrictions on the application developer. While the prototype is not a perfect solution to the problem, it does demonstrate a novel approach to resource control which is very versatile, by nature of being entirely composed of embedded DSLs. It also offers users more insight into what sensitive resources are used for, compared to currently widespread mobile platforms.

In the future, it would be interesting to explore the use of other Racket libraries, particularly Typed Racket, in the hope that we can achieve more reliable restrictions than currently possible. This might be an avenue to pursue in response to the vulnerability that the current prototype has, arising from evaluation of dynamically constructed expressions or allowing module importing.

Acknowledgements

Special thanks go to Ludovic Courtès, Camille Mañano, Andreas Enge, and Hamish Ivey-Law, who proofread early drafts of this work and provided invaluable comments. The constructive criticism provided by the anonymous reviewers is equally appreciated.

8. REFERENCES

[1] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of Sense/Compute/Control applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 431–440, New York, NY, USA, 2011. ACM.

[2] D. Cassou, J. Bruneau, C. Consel, and E. Balland. Toward a tool-based development methodology for pervasive computing applications. *IEEE Trans. Software Eng.*, 38(6):1445–1463, 2012.

[3] Chrome developers. Developing Chrome extensions: Declare permissions. https://developer.chrome.com/extensions/declare_permissions, 2015. Accessed 2/2015.

[4] J. L. Dave Mark. *Beginning iPhone Development: Exploring the iPhone SDK*. Apress, 2009.

[5] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *ACM SIGPLAN Notices*, volume 46, pages 215–226. ACM, 2011.

[6] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang. A static assurance analysis of Android applications. *Virginia Polytechnic Institute and State University, Tech. Rep*, 2013.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[8] J. Feiler. *How to Do Everything: Facebook Applications*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 2008.

[9] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[10] M. Flatt. Submodules in Racket: You want it when, again? In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 13–22, New York, NY, USA, 2013. ACM.

[11] M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that work together. *Journal of Functional Programming*, 22:181–216, 3 2012.

[12] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[13] C. Gibler, J. Crussel, J. Erickson, and H. Chen. AndroidLeaks: Detecting privacy leaks in Android applications. Technical report, Tech. rep., UC Davis, 2011.

[14] R. N. Horspool and N. Tillmann. *TouchDevelop: Programming on the Go*. The Expert's Voice. Apress, 3rd edition, 2013. available at https://www.touchdevelop.com/docs/book.

[15] C. Mann and A. Starostin. A framework for static detection of privacy leaks in Android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[16] R. Rogers, J. Lombardo, Z. Mednieks, and B. Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly, Beijing, 2009.

[17] M. Snoyman. *Developing Web Applications with Haskell and Yesod – Safety-Driven Web Development*. O'Reilly, 2012.

[18] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in Android ad libraries, 2012.

[19] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[20] S. Tobin-Hochstadt and M. Flatt. Advanced macrology and the implementation of Typed Scheme. In *In Proc. 8th Workshop on Scheme and Functional Programming*, pages 1–14. ACM Press, 2007.

[21] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[22] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the Android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2012.

[23] X. Xiao, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal. User-aware privacy control via extended static information-flow analysis. In M. Goedicke, T. Menzies, and M. Saeki, editors, *ASE*, pages 80–89. ACM, 2012.