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 | when (required | provided (srcName | ctxName) |
| GetData | do | publish |
| PublishSpec | (always | maybe) |
| Controller | controller ctrName CtxtInteract |
| 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 to 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 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

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.
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.

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

```
#lang s-exp "webcamspec.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 6: The implementation of the `ComposeDisplay` context.

```
#lang s-exp "webcamimpl.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
    )
  )
)
```

```
; ... ; the remaining implement-terms
```

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

```
#lang s-exp "webcamspec.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 6: The implementation of the `ComposeDisplay` context.

```
#lang s-exp "webcamimpl.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
    )
  )
)
```

```
; ... ; the remaining implement-terms
```

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
implementation should be a lambda term which obeys the contract resulting from the specification. For example, the CompositionDisplay context has the contract \((\rightarrow \text{bitmap}\?) \ (\rightarrow \text{string}\?) \ (\rightarrow \text{bitmap}\? \ \text{void}\?) \ (\rightarrow \text{void}\?) \ \text{none/c})\) This is because it is activated by ProcessPicture publishing an image, it has \text{get-access-to the MakeAd} component which returns a string, and it may optionally publish an image on account of its \text{maybe-publish} specification. The last two arguments correspond to publishing \((\rightarrow \text{bitmap}\? \ \text{void}\?)\) and not publishing \((\rightarrow \text{void}\?)\) 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 conditioned pathways, the \text{implement} macro wraps each \text{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 \text{webcamspec.rkt} language – the one arising from the specifications we have written. The implementation module is checked before run-time to contain exactly one \text{implement} \text{C ...} term for each declared \text{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 \text{(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 \text{#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 \text{#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, e.g., 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 CompositionDisplay 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 \text{implement x ...} terms into \text{(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 \text{#lang s-exp "webcamspec.rkt"} directive.

Line 1 marks the start of the implementation module, called
This code corresponds to the blue box in Fig. 4.

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

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.

---

Tested using DrRacket v6.1.1.
We therefore inspect a developer’s implementation for such vulnerabilities, which in our case could allow them to manipulate sensitive resources. As a simple example, we failed to get an existing communication channel 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 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 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 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.

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


