

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

Paul van Der Walt

▶ **To cite this version:**

Paul van Der Walt. A language-independent methodology for compiling declarations into open platform frameworks. Programming Languages [cs.PL]. Université de Bordeaux, 2015. English. NNT : 2015BORD0288 . tel-01251882v1

HAL Id: tel-01251882

https://inria.hal.science/tel-01251882v1

Submitted on 6 Jan 2016 (v1), last revised 3 Mar 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

This page intentionally left blank to compensate for the title page that HAL prepends. Without this page, the recto/verso layout assumptions are broken.

université de BORDEAUX

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux
Numéro National : 2015BORD0288

Compilation de déclarations dans des cadriciels : une méthodologie indépendante du langage

THÈSE

soutenue le 14 décembre 2015

pour l'obtention du

Doctorat de l'Université de Bordeaux
(spécialité Informatique)

par

Paul van der Walt

Jury

Rapporteurs :	Philippe Lalanda,	Professeur à l'Université Joseph Fourier de Grenoble
	Romain Rouvoy,	Maître de conférences (HDR) à l'Université de Lille 1
Examinateurs :	Charles Consel,	Professeur à l'Institut Polytechnique de Bordeaux
	Nic Volanschi,	Advanced research position à Inria Bordeaux

PAUL VAN DER WALT

A LANGUAGE-INDEPENDENT METHODOLOGY FOR COMPILING DECLARATIONS INTO OPEN PLATFORM FRAMEWORKS

INRIA BORDEAUX SUD-OUEST, FRANCE

LaBRI
Unité Mixte de Recherche CNRS (UMR 5800)
351 cours de la Libération
33405 Talence Cedex
France

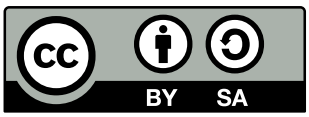
Équipe PHOENIX, INRIA Bordeaux Sud-Ouest
200 avenue de la Vieille Tour
33405 Talence Cedex
France

Université de Bordeaux

Copyright © 2015 by Paul van der Walt

WWW.DENKNERD.ORG

The typographic style of this document was inspired by Edward Tufte's book *Beautiful Evidence*, and typeset using LaTeX and a modified version of Kevin Godby's `tufte-book` class. The main text is typeset in *TeX Gyre Pagella*, which is based on Hermann Zapf's beautiful Palatino type face. The `typewriter` text is typeset in *Bera Mono*, originally developed by Bitstream, Inc.



This work and associated source code is licensed under a *Creative Commons Attribution-ShareAlike 4.0 International License*, available at `https://creativecommons.org/licenses/by-sa/4.0/`.

Might the fleas of a thousand camels descend upon the armpits of those who would dare to make unauthorised copies of this work, in whole or part, without proper attribution. Sickness and ruin upon those who would attempt to derive financial gain from this work, even unto the seventh generation.

Please mind the trees: think before you reproduce.

Version: 6th January 2016.

Abstract

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

In the domain of open platforms, it has become common to use application programming frameworks extended with declarations that express permissions of applications. This is a natural reaction to ever more widespread adoption of mobile and pervasive computing devices. Their wide adoption raises privacy and safety concerns for users, as a result of the increasing number of sensitive resources a user is sharing with non-certified third-party application developers. However, the approach to designing these declaration languages and the frameworks that enforce their requirements is often ad hoc, and limited to a specific combination of application domain and programming language. Moreover, most widely used frameworks fail to address serious privacy leaks, and, crucially, do not provide the user with insight into application behaviour.

This dissertation presents a generalised methodology for developing declaration-driven frameworks in a wide spectrum of host programming languages. We show that rich declaration languages, which express modularity, resource permissions and application control flow, can be compiled into frameworks that provide strong guarantees to end users. Compared to other declaration-driven frameworks, our methodology provides guidance to the application developer based on the specifications, and clear insight to the end user regarding the use of their private resources.

Contrary to previous work, the methodology we propose does not depend on a specific host language, or even on a specific programming paradigm. We demonstrate how to implement declaration-driven frameworks in languages with static type systems, completely dynamic languages, object-oriented languages, or functional languages. The efficacy of our approach is shown through prototypes in the domain of mobile computing, implemented in two widely differing host programming languages, demonstrating the generality of our approach.

Keywords: programming frameworks, domain-specific languages, declaration languages, generative programming, privacy

Résumé

Compilation de déclarations dans des cadriciels : une méthodologie indépendante du langage

Dans le domaine des plates-formes ouvertes, l'utilisation des cadriciels (*frameworks*) enrichis par des déclarations pour exprimer les permissions de l'application est de plus en plus répandue. Ceci est une réaction logique au fait qu'il y a une explosion d'adoption des appareils embarqués et mobiles. Leur omniprésence dans notre vie quotidienne engendre des craintes liées à la sécurité et à la vie privée, car l'usager partage de plus en plus ses données et ressources privées avec des tiers qui développent des applications auxquelles on n'a pas de raison de faire confiance. Malheureusement, la manière dont ces langages de spécification ainsi que ces cadres d'applications sont développés est généralement assez *ad hoc* et repose sur un domaine d'application et un langage de programmation fixes. De plus, ces cadriciels ne sont pas assez restrictifs pour régler le problème de la fuite de données privées et ne donnent souvent pas non plus assez d'informations à l'usager sur le comportement attendu de l'application.

Cette thèse présente une méthodologie généraliste pour développer des cadriciels dirigés par des déclarations, qui cible un spectre large de langages de programmation. Nous montrons comment des langages de déclaration expressifs permettent de spécifier avec modularité les droits d'accès aux ressources ainsi que le flux de contrôle d'une telle application. Ces langages peuvent ensuite être compilés en un cadriciel garantissant à l'usager final le respect de ces permissions.

Par rapport aux cadriciels existants, notre méthodologie permet de guider la personne qui développe des applications à partir des spécifications ainsi que d'informer l'usager final sur l'usage des ressources sensibles. Contrairement aux travaux existants, la méthodologie présentée dans cette thèse ne repose par sur un langage de programmation particulier. Nous montrons comment mettre en œuvre de tels cadriciels dans un spectre de langages : des langages avec typage statique ou dynamique, et suivant le paradigme objet ou fonctionnel. L'efficacité de l'approche est montrée à travers des prototypes dans le domaine des applications mobiles dans deux langages très différents, à savoir Java et Racket, ce qui montre la généralité de notre approche.

Mots clés : cadriciels, langage dédié, langages de déclaration, programmation générative, vie privée

Acknowledgements

In browsing the dissertations of those who have gone before me (when i[1] should have been writing my own manuscript), it struck me that a constant factor across the sciences is the Acknowledgements section devolving into smug yet maudlin name-dropping. I will not disappoint in this regard—i temporarily beg your pardon, beautiful and wise reader, for my self-indulgence. I, too, fall prey to the twin vices of fawning and vanity.[2]

George Orwell could be said to have been constantly relevant to me during my work. In his 1946 essay, *Why I Write*, he describes his process in a way that resonates with me:

> Writing a book is a horrible, exhausting struggle, like a long bout of some painful illness. One would never undertake such a thing if one were not driven on by some demon whom one can neither resist nor understand.

Still, i can only assume that writing a book is infinitely more tiresome than completing a dissertation. My potentially inappropriate feeling of appreciation for Orwell's struggle aside, those devices which almost everyone these days seems to have—devices equipped with microphones, multiple cameras, geographical tracking as well as remote control capabilities—all bear a striking resemblance to the description that Orwell gives of 'telescreens' in *1984*, his famous dystopian novel. Of course, those were merely affixed to the salon wall; today, we carry our battery-powered telescreens around with us wherever we go.

I jest. Obviously, my intellectual debt to Orwell pales in comparison to the amount of gratitude i have for the people who actually enabled my work in practical ways. First and foremost, i thank my doctoral supervisors, Charles Consel and Emilie Balland. Clearly, if it were not for the trust they placed in me by accepting me into the programme, and the guidance they provided me along the way, i would never be where i am today. I would also like to thank my examiners, Philippe Lalanda and Romain Rouvoy, who went to the trouble of reviewing this dissertation, providing constructive criticism, and attending my defence. I extend my thanks to Denis Barthou, who graciously accepted the invitation to serve on my thesis committee. Many thanks also to Nic Volanschi, who replaced him when he fell ill at the last minute! I also express my sincere appreciation for all my friends and colleagues who provided ruthlessly thorough comments on draft versions of my work. These

[1] My uncapitalised usage of the singular first person personal pronoun follows that of danah boyd, to whose philosophy on the subject i subscribe (see http://www.danah.org/name.html).

[2] Please do not misconstrue this good-natured fun-pokage as derision: i understand and have utmost respect for the circumstances and feelings that precipitate such outpourings of sentimentality. Indeed, i have the same sincere desire to thank my entourage!

people include, in no particular order, Damien C., Hamish, Ludovic, Andreas, David R., Milan, Nic and Julien B. This dissertation would certainly never have reached any semblance of acceptability if it were not for your tireless reading and rereading and re-rereading. Of course, the remaining errors are all mine.

Furthermore, to the colleagues who were there for me in my deepest moments of doubt and despair. I cannot express how thankful i am to all of you for being there when i needed you, and for talking sense into me when i was hell-bent on calling it quits. Hervé ("refugees welcome"), David S., Marc, Cédric, Isabelle, Laetitia, Charles F., Lucile, Adrien: without your material and moral support, i would be a very bitter and perhaps unemployed person right now. I would also like to thank Luce, Laure, and Catherine, for making my administrative nightmares go away.

I would also like to mention the people who, years ago, put me on the path that led me here. My teachers at Utrecht University instilled in me a love and fascination for functional programming through their inspiring teaching and general coolness. Especially Wouter, Doaitse, Andres, Johan J., Atze, Jeroen F., Jurriaan: dank jullie wel, jullie waren allemaal een ware inspiratie voor mij! Moge ik ooit ook zo'n goede leraar worden!

Meandering from the professional to the personal, a number of my friends seem to have maintained a limitless tolerance for my foibles and vices, an inexplicable appreciation of my boorish society. You've managed to keep me company even when i emigrated, became physically and/or emotionally distant as a result of my work; even my incessant complaining about life, and my proclivity for poor wit were insufficient to keep you lot at bay. Every time you've welcomed me back with open arms, received me as if i'd never left. Hamish, Koen, Floor, Barinder, Hanna, Andreas, Marleen, Ludovic, Jorrit, Tim, Dieuwertje, Tristan, Chloë, Linde, Maja, Benjamin, Justin, Gabe, Mai, Twan (in `sort -R` order). Thank you! I very much owe all of you a multitude of beers. To those of you who came to visit me in Bordeaux, rest assured that your geographically optimised companionship was much appreciated. Clearly my folks, Ina and Will, also deserve special mention. It is you who put me on this ball of mud, and in spite of what an ungrateful sod i can be still occasionally provide me with a place in the Netherlands to crash! Thank you!

Finally, last but most certainly not least: Camille, *merci de m'avoir supporté,* and thank you for having supported me. You showed infinite patience when i was scared or super stressed out, provided me with a shoulder to lean on, and you encouraged me to keep on keeping on when things seemed unbearable. It is you who made late-stage Bordeaux tolerable, who gave me an excuse not to give up every day. Also, don't feel bad that you were left till the end—it is well-established that to be thanked last is the highest honour.[3]

[3] Ken Hyland (2003). 'Dissertation Acknowledgements: The Anatomy of a Cinderella Genre'. In: *Written Communication* 20.3, pp. 242–268.

Résumé étendu

Cette thèse, intitulée « Compilation de déclarations dans des cadriciels : une méthodologie indépendante du langage », est située dans le champ du génie logiciel et de l'informatique mobile. Plus précisement, ce travail de thèse se situe dans le domaine des applications pour des appareils mobiles et de la gestion des droits d'accès aux ressources. Il propose une méthodologie de développement fondée sur les notions de plates-formes ouvertes, de déclarations et de compilation pour la création de cadriciels (*frameworks*) dédiés.

Problématique. Une *plate-forme ouverte* est une plate-forme informatique qui propose aux applications (1) une interface de programmation publique (API), (2) l'accès à des ressources partagées, (3) un répertoire où des développeurs d'applications tiers peuvent proposer pour installation leurs applications aux utilisateurs finaux. Les exemples dans la vie quotidienne de ce type de plates-formes sont nombreux, notamment Android ou Apple iOS. On compte de plus en plus d'usagers de ce type de plate-forme, leur confiant de plus en plus des données privées à stocker, alors que l'on est incapable de bien gérer les droits des applications auxquelles on ne fait préalablement pas confiance et qui ne sont pas certifiées. Ceci rend possible notamment des fuites de données privées ou sensibles, comme par exemple des messages privées, courriels, images, informations sur la géolocalisation de l'utilisateur, historique des appels, *etc.*

On illustre la problématique et notre approche à l'aide d'une application mobile basée sur des applications répandus. L'application prétend ne faire que prendre une photo avec la caméra de l'appareil mobile, et la modifier ensuite avec un filtre d'image. Cette application est gratuite et montre des publicités aux usagers, ce qui nécessite d'accéder au réseau. Or, l'application malveillante n'utilise pas ces privilèges seulement à bon escient ; elle arrive à exfiltrer des données privées, qui justement ont été rendues accessibles pour le bon fonctionnement de l'application. Ce type d'attaque contre la vie privée a été déjà fréquemment signalé pour des applications répandues, et la crainte est que la nombre croissant des appareils mobiles va rendre de plus en plus pertinent ce problème.

Contexte du travail, contributions. Ce travail reprend les fondements scientifiques de l'outil DiaSuite,[4] précédemment développé dans

[4] Plus d'informations sont disponibles sur http://phoenix.inria.fr/research-projects/diasuite.

le même groupe de recherche, afin de répondre à cette problématique. Le point fort dans ce contexte de l'approche DiaSuite est le découpage des applications en composants, et l'explicitation des interactions entre eux. Ceci nous permet d'avoir une vision plus claire du flux des données. En plus de donner à l'usager une vision de la structure et du comportement de l'application, cette approche permet de restreindre ce dernier en utilisant les déclarations comme point d'appui pour générer des cadriciels.

Les contributions principales de ce travail de thèse peuvent être catégorisées en deux aspects : la formalisation des fondements de l'approche DiaSuite, et le fait qu'on montre que cette méthodologie s'applique (1) sur un vaste spectre de paradigmes de langages de programmation (typage dynamique, typage statique, impératif, fonctionnel) et (2) sur la problématique de la vie privée des usagers dans les systèmes mobiles. Ceci constitue une avancée par rapport à l'état de l'art existant. En outre, les travaux sur les fondements des cadriciels dans le contexte d'un langage fonctionnel apportent une réflexion nouvelle. Le plus souvent, on retrouve des discussions sur des cadriciels réalisés dans les langages typés, et construits à partir des concepts orienté-objet. Le développeur doit implémenter son application par extension des objets fournis par le cadriciel, ce qui assure la bonne structure de l'application. Le respect des contrats définis est ensuite pris en compte par des systèmes de typage et par l'extension et l'implémentation des objets.

En revanche, notre travail montre que c'est également intéressant et faisable d'étudier la problématique des cadriciels d'un point de vue syntactique, et basé sur des concepts des langages de programmation et, de surcroît, dans le contexte des langages fonctionnels. Enfin, ce travail propose l'approche novatrice d'un langage dédié à la spécification qui lui-même, une fois évalué, constitue une langage dédié et adapté à l'implémentation spécifique.

Structure du document. Le manuscrit est organisé de la façon suivante. Un premier chapitre introductif situe le contexte du travail. La notion de plate-forme ouverte pour le développement et l'exécution d'applications mobiles est brièvement introduite ; elle sera détaillée dans les chapitres suivants. Les droits d'accès aux ressources, y compris *via* Internet, sont également présentés. Cette problématique constitue un enjeu majeur pour l'acceptation des applications mobiles et justifie le travail de thèse. Enfin, ce chapitre définit l'objectif du travail : l'identification de principes généraux pour le développement de plates-formes ouvertes utilisant des déclarations, notamment pour gérer les droits d'accès aux ressources. Le premier chapitre définit également des exigences à propos de ces plates-formes ouvertes. Ces exigences sont définies en regardant les plates-formes existantes et répandues, ainsi que le travail plus théorique qui a été fait par Consel et Balland (2010). Elles seront ensuite utilisées pour évaluer le travail de la thèse.

Le deuxième chapitre fournit des éléments de contexte et de

positionnement. Deux points essentiels sont abordés. Tout d'abord, il est rappelé que deux approches sont mises en place pour vérifier le respect de la vie privée : les approches *statiques* où le code est analysé avant déploiement et exécution, et les approches *dynamiques* où le code est analysé à l'exécution. Les avantages et limites de ces deux approches sont synthétisés. Le deuxième point traité par ce chapitre est la notion de plate-forme ouverte. Pour cela, la définition de Balland et Consel (2010) est reprise et développée. Il met également en avant l'utilisation de déclarations au sein de ces plates-formes. C'est notamment le cas de DiaSuite, fondé sur le motif Sense/Compute/Control définit par Taylor et al. (2009).

La seconde partie de ce manuscrit se concentre sur la contribution de thèse. Elle est détaillée avec précision dans le chapitre 3. La contribution se situe à deux niveaux :

1. l'utilisation du motif Sense/Compute/Control pour le développement d'applications mobiles, et notamment à partir de l'outil DiaSuite. À ce jour, cet outil et l'approche afférente étaient utilisés pour les applications pervasives utilisant des capteurs issus de l'environnement physique. Ici, les ressources des équipements mobiles sont considérées et gérées comme des capteurs.

2. une méthodologie pour le développement de plates-formes ouvertes fondées sur la notion de déclaration. Cette méthodologie, générale, se focalise surtout sur le processus de compilation qui génère des canevas de développement adaptés aux applications préalablement spécifiées (et fondés sur le motif Sense/Compute/Control précédemment mentionné).

Le chapitre 4 apporte un support formel à la proposition. C'est le chapitre central de cette thèse. Il présente l'architecture du compilateur de déclarations, définit le système de type utilisé, et spécifie la sémantique de la principale phase de compilation. La motivation pour reprendre les fondamentaux de DiaSuite est que le travail précédent à été fait d'une façon pragmatique, et surtout que des décisions de conception on été prises en fonction de langage d'implémentation qui a été utilisé, c'est-à-dire Java. Notre travail sépare la méthodologie du langage d'implémentation, et on montre le spectre des possibilités pour des décisions de conception.

Le quatrième chapitre reprend également les exigences posées dans le chapitre introductif et positionne la proposition par rapport à ces exigences. Un point majeur est la finesse des déclarations : dans l'approche proposée, les droits d'accès aux ressources sont spécifiés au niveau des composants et de leurs interactions. Cela permet aux développeurs de spécifier les interactions entre composants et les droits associés. Le choix est ainsi de vérifier le respect des permissions de façon statique, mais on remarque également que ce choix n'est pas la seule possibilité. Le chapitre comprend une discussion des avantages et inconvénients des choix à propos de la vérification des permissions statique ou dynamique, et donne des conseils aux futurs implémenteurs selon les besoins spécifiques.

Les chapitres 5 et 6 se focalisent sur l'implémentation de l'approche. Deux langages cibles ont été choisis dans un souci de validation : Java et Racket. Java est un exemple de langage typé statiquement et orienté objet, alors que Racket représente un langage fonctionnel à typage dynamique. Ces deux chapitres reprennent la même organisation : une présentation générale de la conception du prototype, la projection des déclarations dans le langage cible, un exemple d'utilisation et une évaluation du prototype au regard des exigences précédemment définies.

Le chapitre sur Racket incarne une contribution nouvelle, montrant qu'il est possible d'apporter un niveau d'aide au développeur Racket comparable à celui fourni avec Java, notamment en ce qui concerne les restrictions d'accès aux ressources. L'implémention des garanties statiques et même des cadriciels en général dans un langage dynamique était jusqu'à aujourd'hui peu étudié.

Ayant implémenté notre méthodologie dans un langage dynamiquement typé, on montre qu'il n'est pas obligatoire d'avoir une système de typage statique pour arriver à fournir des garantis comme prévu dans la méthodologie. Le contraire avait été supposé auparavant dans la thèse de Cassou (2011).

Les chapitres 8, 9 et 10 concluent le manuscrit. Le chapitre 8 avance le fait que l'approche proposée est également applicable au domaine de l'informatique ubiquiste dans la maison (*assisted living* en anglais). Un exemple d'une tel application est tiré des travaux récents de Caroux (2014).

Le chapitre 9 reprend des éléments liés au positionnement des travaux de cette thèse, notamment en ce qui concerne la gestion des accès aux ressources privées. Enfin, le chapitre 10 développe des perspectives de ce travail. Même si les prototypes présentés dans ce travail ont des limitations, les perspectives sont encourageantes notamment en envisageant la combinaison de cette méthodologie avec des techniques existantes comme dans la domaine du *component-based software engineering*, qui peuvent servir des composants assurant la fonctionnement dépendable d'une application, même en ajoutant des facilités pour l'usage des bibliothèques externes.

Le code source des prototypes dévéloppés dans le cadre de cette thèse est disponible en ligne.[5]

[5] Le code source est téléchargeable depuis le site Web `http://people.bordeaux.inria.fr/pwalt/`

Publications scientifiques. Les travaux présentés dans cette thèse ont donné lieu à un article publié dans une conférence internationale :

- Paul van der Walt (2015). 'Constraining application behaviour by generating languages'. In: *8th European Lisp Symposium*. London, United Kingdom.

Un autre article a été soumis dans la revue scientifique *Software: Practice and Experience* :

- Paul van der Walt, Charles Consel and Emilie Balland (2015). 'Frameworks compiled from declarations: a language-independent approach'. manuscrit.

Contents

I Context

1 *Introduction* 3

1.1 Requirements 4

1.2 Contributions 6

1.3 Outline 7

2 *Context and Preliminaries* 9

2.1 Program analysis as a privacy measure 10

2.2 Libraries, frameworks and their evolution 12

2.3 Open platforms 13

2.4 Previous work on DiaSuite 16

II Presenting the Methodology

3 *Software development with tailored frameworks* 19

3.1 Sense/Compute/Control and mobile computing 19

3.2 Core DiaSpec declaration language 22

3.3 Phases of application development 24

4 *Semantics of the declarations* 31

4.1 Phases of the declaration compiler 31

4.2 Core DiaSpec: typing rules 34

4.3 The framework construction phase of the compiler 40

4.4 Static vs. *dynamic checks* 43

4.5 Discussion 44

III Implementations

5 Instantiation of the methodology in Java 49
5.1 Overview of the implementation 50
5.2 Translation of the declarations 51
5.3 Implementing the example application 53
5.4 Evaluation of conformance to requirements 56

6 Instantiation of the methodology in Racket 59
6.1 Overview of the implementation 60
6.2 Translation of the declarations 61
6.3 Implementing the example application 64
6.4 The framework and run-time 66
6.5 Evaluation of conformance to requirements 69

7 Lessons learned 75
7.1 Comparison to mainstream frameworks 75
7.2 Principles 76

IV Conclusion

8 Generalisation of our approach 83
8.1 Application to assisted living 83
8.2 Other application domains 84

9 Related work 85
9.1 Frameworks enriched with declaration languages 85
9.2 Security of Android applications 86
9.3 Operating system security 87
9.4 Language-level restrictions 87

10 Conclusion and perspectives 91
10.1 Assessment 92
10.2 Ongoing and future work 93

A Code listings 99

Bibliography 105

List of Figures

1 The *Sense/Compute/Control* paradigm. Illustration adapted from Cassou, Bruneau et al. 2012. 15

2 The SCC model as applied to the domain of mobile computing. 20
3 Simplified schematic of the design of the example application. We do not want the picture to be able to leak to the Internet. The application developer provides the components indicated by the grey ovals, the dotted grey square indicates the boundary between the platform and the application. 21
4 The grammar of Core DiaSpec, our specification language. It will serve as an example in the rest of this work. Keywords are in bold, terminals in italic, and rules in normal font. 22
5 The Core DiaSpec specification for our example image filter application. 24
6 The phases of development, from specification of an application through to execution. 25
7 The taxonomy of devices for our mobile computing example. 26

8 The global phase diagram for the specification compiler. It outputs the framework intended to be used for implementing the application. The type checking and framework construction phases are discussed in detail in this chapter. 32
9 The production rules for the type environment and its terms. 35
10 The typing rules for source and action terms. The side condition unique? ensures that bindings are not shadowed in the environment Γ. 35
11 A few representative examples of typing rules for context declarations. 36
12 The typing rule for controller declarations. 37
13 The typing judgement check-spec checks a specification, which is a list of declarations, by tail recursion. 38

14 Schematic design of the Java prototype. Note the derived names in the generated abstract class, top right. 51
15 The implementation of the `ComposeDisplay` context. 54
16 Screenshot of the Eclipse Java IDE, showing tailored suggestions based on declarations. 54
17 Excerpt of `AbstractComposeDisplay`, including the `MakeAd` proxy. 55

18 Excerpt of `AbstractMakeAd` showing the inner proxy class, which gives access to the Internet. 56
19 Example of deployment and binding of implementations to names from the specification. All but one of the instantiations have been elided. 56

20 The architecture of the prototype. Provided declarations are transformed into a tailored language for the implementation. The `implement` macro gets cases for each declared component. The square brackets indicate the result of module expansion. 60
21 Complete declarations of the example application, in Racket prototype. 61
22 The implementation of the `ComposeDisplay` context. The developer creates a new canvas and paints the received image `pic` onto it, followed by the text `adTxt`. 64
23 Separation of components using modules. The developer's code (left), and its expanded form (right). The function f in C cannot access D or g, because of lexical scoping. 65
24 The developer's code snippet is encapsulated in a submodule, as a result of evaluating Figure 22. The shaded code is simply the term provided by the developer, which has been spliced into a new submodule. 65
25 The simplified expansion of the specifications, concentrating on `ComposeDisplay` from Figure 21. 67
26 Unmodified screenshot of the DrRacket GUI, indicating available binding information. Note the blue arrow in the code window, pointing from the binding site of the `ProcessPicture` term (line 10) to where it is used (line 16). 69

27 Complete grammar of the Core DiaSpec specification language as modelled in PLT Redex, extended with type environments. 99
28 The complete list of type judgements for Core DiaSpec specifications. 100
29 Definitions of the metafunctions used in the type judgements. 102
30 The option type, `Maybe<T>`. Implemented as 3 separate classes. 103
31 Helper macro to translate terms of the form `(implement x ...)` into `(implement-x ...)`. 104
32 Snippet spliced into all specification modules at expansion time. The **`#%module-begin`** macro allows the specification file to be used as a Racket language extension. 104

Part I

Context

1
Introduction

Within the last 10 years, there has been an explosion in the number of embedded and mobile computing devices pervading our environments.[1] Especially smartphone applications and assisted living are two booming open platform domains which bring new challenges for application developers and platform owners alike.[2] For example, end users routinely install applications on their mobile phones from untrusted sources, possibly giving the application access to private resources, whether these be hardware or software. Potentially sensitive information needs to be accessible to applications for them to be able to achieve their purpose. Therefore, a trade-off must be found between ensuring the privacy of the user on the one hand, and allowing useful and legitimate applications to be created on the other.

[1] William Enck (2011). 'Defending Users against Smartphone Apps: Techniques and Future Directions'. In: *Information Systems Security*. Ed. by Sushil Jajodia and Chandan Mazumdar. Vol. 7093. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 49–70.

[2] Emilie Balland and Charles Consel (2010). 'Open Platforms: New Challenges for Software Engineering'. In: *Programming Support Innovations for Emerging Distributed Applications*. PSI EtA '10. Reno, Nevada: ACM, 3:1–3:4.

In reaction to these challenges, platform owners have introduced various forms of permission declaration languages. By restricting the list of permissions that is granted to a given application, the run-time system can ensure that invalid access to shared resources is blocked. Nevertheless, considering the Android platform for example, numerous studies[3,4,5] and user stories show that there are still gaping holes in the security of these systems—user privacy is routinely breached for purposes ranging from tracking by advertising networks through to downright malicious applications intending to steal private information for the purpose of identity theft.

[3] Quang Do, Ben Martini and Kim-Kwang Raymond Choo (2015). 'Exfiltrating data from Android devices'. In: *Computers & Security* 48, pp. 74–91.

[4] Ryan Stevens et al. (2012). 'Investigating User Privacy in Android Ad Libraries'. In: *Workshop on Mobile Security Technologies (MoST)*

[5] Xuetao Wei et al. (2012). 'Permission Evolution in the Android Ecosystem'. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida, USA: ACM, pp. 31–40.

We observe that in the very popular platforms such as Facebook[6] and Android,[7] which are used by millions of users daily, users are presented with a list of permissions an application requests before it is run for the first time. On the face of it this seems like a reasonable first step towards user privacy. However, numerous shortcomings quickly emerge. Firstly, users do not always understand the privacy implications of various permissions,[8] and get into the habit of always accepting whatever the application demands. Secondly, the permissions are coarse-grained—*e.g.*, access to the entire external storage (SD) card is requested, instead of a particular

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

[7] Ed Burnette (2009). *Hello, Android: Introducing Google's Mobile Development Platform*. 2nd edition. Pragmatic Bookshelf.

[8] Adrienne Porter Felt et al. (2012). 'Android permissions: User attention, comprehension, and behavior'. In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, p. 3.

directory—and apply to the entire application. This is especially troublesome since by default, included advertisement libraries run in the same process as the host application, and therefore have the same permissions as the host application. This has lead to several documented privacy breaches.[9]

Finally, permissions have to be accepted on an all-or-nothing basis, while there is no fundamental reason for this. We note that a recent version of Android[10] includes a hidden screen to (dis)allow access per-application and per-resource. This change highlights that the old permission model was not meeting users' needs. On Apple's iOS this problem is solved by dynamic permissions: a user is queried per-application the first time a sensitive resource is accessed. However, this still does not prevent a malicious application from exfiltrating data after a legitimate request. To curb leaks of sensitive information, we therefore consider the possibilities provided by enriched declaration languages providing finer-grained permission controls plus restrictions on application control flow.

We also note that most frameworks used in production environments make use of specialised techniques for ensuring non-malicious behaviour of the application. The features provided for this as well as the various ways they are implemented give us the impression that their solutions are ad hoc and narrow.

[9] Stevens et al. 2012; and Wei et al. 2012

[10] Users discovered that in version 4.3 of Android, a configuration screen is available which allows selectively granting an application a subset of its requested permissions—*e.g.*, GPS accessible but contact list forbidden. This panel had not been made accessible via the normal GUI. See *Hidden Android feature allows users to fine tune app permissions* (2013). Online, http://www.zdnet.com/hidden-android-feature-allows-users-to-fine-tune-app-permissions-7000018944/. Accessed: May 2015.

Our research question is whether there are underlying principles for designing declaration-driven frameworks which apply to all host languages. Also, we investigate what features are indispensable for a host language to be able to support such a system. This question is inspired by the suggestions for future work made by Cassou;[11] little work has yet been done in this direction.

Additionally, widespread programming frameworks used to develop applications for open platforms lack mechanisms for supporting the developer. An application developer must provide the aforementioned permission declarations, but when implementing the application is still faced with the same level of complexity as with traditional frameworks. We demonstrate that the declarations can also be used to offer implementation guidance to the application developer, and if presented in a comprehensible fashion, can offer end users more insight into the behaviour of the application, thus empowering both end users and developers alike. These advantages hold in a broader context than previously assumed.

[11] Damien Cassou (2011). 'Développement logiciel orienté paradigme de conception : la programmation dirigée par la spécification'. PhD thesis. Université Sciences et Technologies–Bordeaux I.

1.1 *Requirements*

The study of existing programming frameworks for open platforms gives us a practical basis for identifying the requirements for declaration-driven frameworks. We now examine those requirements.

Behavioural transparency. First and foremost, since our goal is to protect the privacy of end users, we believe that users of an open platform should be informed of what will be done with their resources before running a given application. Currently, end users are accustomed to downloading and running applications from unknown third-party developers, which provide either no information about their behaviour at all[12] or merely provide a list of permissions that they will use—for example, the internet connection or file-system access—that a user has to accept or reject wholesale.[13] These permission systems provide little to no insight into what an application intends to do with the user's private data. The clear presentation of permissions is also an important requirement. Permission declarations are of no use if the user must be an expert of the platform to be able to understand their impact.[14]

[12] As in the case of applications on general purpose computing platforms such as Linux or OS X.
[13] As is the case on the Android platform, as well as numerous others such as Facebook or Chrome plugins; see Rogers et al. 2009, Feiler 2008 and Chrome developers 2015 respectively.

[14] Felt et al. 2012

Coherence between specification and behaviour. For this methodology to be effective, an application should be guaranteed to adhere to the specifications provided by the developer and presented to the user.[15] In our setting, that means that if a user approves a certain set of permissions for a given application, they should be sure in the knowledge that the application will not be able to circumvent the restrictions. In essence, this requirement implies a strong semantic link between the specification, the implementation, and the run-time environment of the application.

[15] Richard N. Taylor, Nenad Medvidovic and Eric M. Dashofy (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing

Development support. Developers of applications should be guided as much as possible to reduce cognitive load as well as to reduce the probability of unintentional bugs. This is a particular advantage identified in the DiaSuite approach by Cassou, Bruneau et al. 2012,[16] which we argue should be available to all application developers. As a valuable additional benefit, if the developer experience is positive, it will mean more high-quality applications available to the end user, which will in turn make the platform more valuable.

[16] Damien Cassou, Julien Bruneau et al. (2012). 'Toward a Tool-Based Development Methodology for Pervasive Computing Applications'. In: *IEEE Trans. Software Eng.* 38.6, pp. 1445–1463

Abstraction over host language. The methodology we propose should not be dependent on a given host language. Particularly, a platform owner who has already invested in a given programming language cannot be expected to abandon it, even for significant benefits in increased user privacy. Therefore, a methodology claiming to solve this problem should be easily applicable to arbitrary target languages. Notably, it should allow implementations in a wide range of programming paradigms, *e.g.*, object-oriented or functional, and it should support implementation in either dynamic languages, or languages with static type systems.

1.2 Contributions

This dissertation proposes an approach that covers the design and development of *application-tailored* declaration-driven frameworks in a wide spectrum of host programming languages, addressing the requirements identified above. The main contributions of this thesis are:

A declaration-driven development approach for open platforms. This work starts with a study of other declaration-driven frameworks. As stated, examples include the Android SDK, Apple's iOS, Chrome SDK, Facebook SDK, DiaSuite, but also more niche frameworks such as Yesod,[17] the statically-checked web framework in Haskell. We compare the approach taken by each of these, and their relative advantages and disadvantages. Also, we go further than previous work, such as Cassou 2011, with regards to the formalisation of the methodology. We detail the compiler architecture involved in a system which generates *application-tailored programming frameworks* from specifications, and formalise key phases. Notably, we provide a type system for our declaration language, and a more systematic specification for the declaration compiler, plus justification why the guarantees provided by our methodology are stronger than mainstream declaration-driven frameworks.

[17] Michael Snoyman (2012). *Developing Web Applications with Haskell and Yesod*. O'Reilly.

A language-independent methodology. The main contribution of this work is showing that our methodology is to a great extent language-agnostic. That is, we provide a systematic way of implementing declaration-driven framework compilers targeting a wide spectrum of programming languages, which fulfil the requirements outlined above. We discuss the design space of compilers generating such frameworks, with regard to the trade-offs between static *vs.* dynamic treatment of declarations, and the minimum requirements for a target programming language to support the methodology.

Case study implementing declaration compilers in Java and Racket. We demonstrate the systematic implementation of framework compilers for our declarative method, targeting two very different languages. We provide compilers targeting Java, an object-oriented and statically typed language, and Racket, a dynamic functional language. We compare the application-tailored frameworks which result from the two compilers, and show that the guarantees provided by both of them are stronger than in widespread declaration-driven frameworks such as Android.

Validation of the methodology. To illustrate and validate our methodology for varying target languages, we implement a simple mobile application in the two presented frameworks. We show that the support and guarantees they offer are equivalent.

1.3 Outline

The remainder of this dissertation is organised as follows:

- Chapter 2 presents the previous work on frameworks and declaration languages as well as all other work which is required to understand what is presented in this dissertation, and on which we will be building. The study of both static and dynamic analysis has a long history, detailing the trade-offs between various approaches which are relevant for our work. More recent work specifically targeting the domain of mobile applications is also discussed here. The Sense/Compute/Control paradigm is also recalled and explained.

- In Chapter 3, the Sense/Compute/Control methodology is instantiated for the domain of mobile computing. We introduce our running example from that domain: a social media application, which is a simplified model of a photo-manipulation application. We introduce our prototype specification language. It is this language which we will use to illustrate our specification-to-framework compiler development methodology. Chapter 4 provides a detailed architecture of the declaration compiler, introduces the type system, and specifies the semantics of the main compilation phase.

- Chapter 5 and Chapter 6 present the implementation of the example application, from the developer's point of view, in the two instances of the framework, respectively Java and Racket. We thoroughly discuss the more pragmatic implementation decisions. Also, we show that both the developer guidance and restriction provided are equivalent, and in accordance with the requirements enumerated above. Chapter 7 concludes the implementation part of this dissertation, and compares our approach with mainstream approaches. We present principles for development of declaration-driven frameworks applicable to other target languages, guided by our experience implementing the prototypes.

- Chapter 8 contains a discussion on the broader applicability of this methodology to other application domains. Not only the domain of mobile computing, but also assisted living, is a prime candidate for application of our methodology.

  Furthermore, we detail an as-yet unfinished experiment towards developing a simulator based on log file replay to augment the development cycle, which we believe will aid the developer to write more dependable applications prior to deployment.

- Finally, Chapter 9 lists other work which is related to ours, and Chapter 10 details the conclusions of this dissertation. Avenues for further work are discussed, such as how to present potential information flow to the end user, so that it is most meaningful to

a non-expert audience. The aim is to help users make informed decisions regarding privacy trade-offs.

2
Context and Preliminaries

This chapter aims to present some history, as well as the context within which this work has been performed. We present all the definitions necessary to understand the rest of this dissertation.

We give a brief overview of the study of both static and dynamic program analysis in Section 2.1. This field has a long history of investigating the trade-offs between various approaches, which remain relevant to our work. However, we concentrate on more recent work specifically relating to privacy and security on mobile platforms. Since it is closely related to the work presented in this dissertation, it deserves a more thorough presentation than program analysis in general.

Apart from static analysis, another approach to user privacy and application safety is to develop more restrictive programming frameworks. We therefore dedicate Section 2.2 to presenting the evolution of software engineering practice from libraries to programming frameworks. Also, we present the recently popular idea of *declaration-driven frameworks*. Our methodology draws heavily on the ideas underpinning widely deployed declaration-driven frameworks.

Section 2.3 gives a definition for the concept of open platforms. Furthermore, we go into more detail on declaration languages, and the challenges and requirements motivating their development, their evolution, as well as the current state of the art. We also introduce the Sense/Compute/Control paradigm, which is the software architecture on which our methodology is based.

Finally, Section 2.4 presents existing work towards enriching declaration languages, using DiaSuite as an experimental vehicle. DiaSuite is a declaration-driven open platform developed in the past by the research group. A later section is devoted to explaining the elements from DiaSuite that we build on. We make reference to specific extensions to DiaSuite, including application specifications which declare Quality of Service and exception handling constraints. The methodology presented in this dissertation attempts to take a step back and generalise what have been ad hoc approaches to solving highly specific problems. Our experiments suggest what the minimum requirements are for a programming language, to be

able to host instantiations of declaration-driven frameworks using our methodology.

2.1 *Program analysis as a privacy measure*

Since the motivation of our work focuses mainly on privacy and safety concerns, the vast field of program analysis should be examined for inspiration and comparison. For many years, program analysis, whether static or dynamic, has attempted to reliably and tractably answer questions for given application software, such as, 'are the permission restrictions respected', or 'does sensitive information leak to unauthorised sinks'. There exist a number of well-known trade-offs when doing program analysis, for example the fact that static analysis allows catching errors earlier, whereas dynamic analysis is more accurate.[1] For example, static analysis of a mobile application might indicate that it requires permission to send an SMS, but dynamic analysis might reveal that the component which sends an SMS is never activated. These issues are considered general knowledge, and will therefore not be addressed at length in this section. We restrict our literature review to work specifically aimed at static analysis of Android permissions, since this comes closest to the domain of our work. However, in the rest of this thesis we consider the implications of various combinations of static and dynamic checking on our specific application domain. Indeed, we remark that one need not decide to use exclusively dynamic or static analysis: a combination is most often the best approach. Choosing the phase in which to perform a given check, however, is where the true subtlety lies.

[1] Flemming Nielson, Hanne R. Nielson and Chris Hankin (1999). *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Static analysis for Android

Much specialised work exists studying the possibilities of static analysis specifically of Android applications. This work aims at doing analysis of existing source code with the constraint that the run-time library remains unchanged. This is motivated by the large body of existing deployed applications that cannot realistically be reengineered (see Elish et al. 2013; C. Mann and Starostin 2012; Fritz et al. 2013; Gibler et al. 2011; Mirzaei et al. 2012).[2] This work is motivated by user privacy and safety concerns. This type of approach has the disadvantage of requiring invasive inspection of the developer's code, and providing no guidance to the developer at implementation time. Unfortunately, application developers are unlikely to want to submit their source code for analysis because of intellectual property concerns. Static analysis approaches that do not inspect developer source code also exist. They convert compiled object code into Java bytecode, for which analysis tools exist.[3] However, this too poses difficulties, since such a decompilation-based approach is frequently impossible or at best inaccurate. Analysis of a program written in a general-purpose language is a very subtle

[2] Karim O. Elish et al. (2013). 'A static assurance analysis of Android applications'. In: *Virginia Polytechnic Institute and State University, Tech. Rep*; Christopher Mann and Artem Starostin (2012). 'A framework for static detection of privacy leaks in Android applications'. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, pp. 1457–1462; Christian Fritz et al. (2013). *Highly Precise Taint Analysis for Android Applications*. Tech. rep. TUD-CS-2013-0113. EC SPRIDE; Clint Gibler et al. (2011). *AndroidLeaks: Detecting Privacy Leaks in Android Applications*. Tech. rep. UC Davis; and Nariman Mirzaei et al. (2012). 'Testing Android Apps Through Symbolic Execution'. In: *SIGSOFT Softw. Eng. Notes* 37.6, pp. 1–5.

[3] S. Holavanalli et al. (2013). 'Flow Permissions for Android'. In: *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pp. 652–657.

problem, since there are many ways in which information flow may be obscured—for example, file-system access, variable aliasing, *etc.* Static analyses are therefore necessarily conservative in these situations.

TouchDevelop: a simplified application creation DSL. We expand our search of related work to include methods which do not necessarily analyse compiled application binaries. Xiao et al. present an application creation environment which aims to restrict sensitive data usage by construction.[4] The main aim of their approach is to give users more insight into data flow, without requiring a full static analysis of a standard Android binary application. This is achieved by providing a domain-specific programming language (DSL) based on TouchDevelop. TouchDevelop is an application creation environment allowing developers to write scripts for mobile devices and publish them in an application store for users to install.[5] It offers an imperative, statically-typed language, but which is not as expressive as normal Java as used with the Android SDK. The authors have developed a static information flow analysis of this DSL, as well as a modified run-time which allows individual resources, such as the contact list, to be replaced by mock components providing anonymised values. The fact that the DSL has limited expressiveness compared to general-purpose Java facilitates static analysis. Furthermore, the advantage is that the user is empowered by per-resource permissions, and comprehensible display of the potential flow of information (*e.g.*, camera → WWW, meaning that data from the camera might be transmitted to the Internet). However, it requires using a separate, specialised application store. Also, regular Android applications are incompatible with the TouchDevelop run-time library. Developers need to learn a new programming language and development environment.

[4] Xusheng Xiao et al. (2012). 'User-aware privacy control via extended static information-flow analysis'. In: *ASE*. ed. by Michael Goedicke, Tim Menzies and Motoshi Saeki. ACM, pp. 80–89.

[5] R. Nigel Horspool and Nikolai Tillmann (2013). *TouchDevelop: Programming on the Go*. 3rd edition. The Expert's Voice. available at `https://www.touchdevelop.com/docs/book`. Apress.

Dynamic analysis: TaintDroid, remote parallel execution

The authors of TaintDroid[6] and Paranoid Android[7] propose another novel approach: real-time, dynamic, taint analysis of applications on a mobile phone, run in parallel on a remote server. This approach is the most accurate of those we compared, but incurs non-negligible costs for platform owners: effectively having to emulate all running user sessions. It illustrates the great accuracy of dynamic analysis, and presents a very interesting experiment. However, a static analysis is more appropriate in settings where CPU power and bandwidth are limited, as is the case in the mobile computing domain. Also, this approach would not scale on the server side, if billions of users' sessions needed to be duplicated remotely. Privacy concerns also arise from the fact that all user actions can be exhaustively tracked and analysed by the platform owner, which merely displaces the trust requirement from the developer to the platform owner. Ideally, the user would have full control over their

[6] William Enck et al. (2014). 'TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones'. In: *Communications of the ACM* 57.3, pp. 99–106

[7] Georgios Portokalidis et al. (2010). 'Paranoid Android: Versatile Protection for Smartphones'. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. Austin, Texas, USA: ACM, pp. 347–356.

sensitive data, and this data should never leave their device unless this is explicitly the intention of the user—*e.g.*, sharing a photo with a friend. In practice, this means that the vast majority of data present on the device should never have to leave it.

2.2 *Libraries, frameworks and their evolution*

Taking a step back from program analysis, we investigate other approaches to user privacy that have been proposed. This section deals with the role that frameworks play in achieving this objective.

Software reuse is agreed to be a goal in itself, for keeping applications maintainable, facilitating the development process, and avoiding repetition.[8] To this end, software libraries have long met a need in software engineering. Going beyond this, the popularity of programming frameworks has been driven by advantages like ease of development, alleviating the burden for developers by managing the execution life cycle of the application, and preventing deviation from the architectural style,[9] while still providing access to shared subroutines. Frameworks are like libraries which exercise authority: instead of a developer writing a whole application from scratch and calling routines provided by a library, a framework takes over and manages the control flow, calling the snippets a developer has provided.[10]

[8] Ruben Prieto-Diaz and Peter Freeman (1987). 'Classifying software for reusability'. In: *Software, IEEE* 4.1, pp. 6–16

[9] Taylor, Medvidovic and Dashofy 2009

[10] Mohamed Fayad and Douglas C. Schmidt (1997). 'Object-oriented application frameworks'. In: *Commun. ACM* 40.10, pp. 32–38.

Their advantages typically include (1) *reducing development effort* by guiding the developer, (2) *restricting* to a particular architectural style, and (3) *fulfilling the needs previously met by libraries* regarding code reuse, in other words, providing easy access to common or shared software artefacts.

One definition for software frameworks found in the literature is 'a collection of several fully or partially implemented components with largely predefined cooperation patterns between them. A framework implements a software architecture for a family of applications with similar characteristics, which are derived by specialisation through application-specific code'.[11]

Pragmatically, programming frameworks are defined by Fayad et al. as a software engineering technique employing *inversion of control*,[12] for creating applications through extension.[13] Contrary to libraries, they can therefore be seen as a technique to turn full application development into a hole-filling activity: the framework provides a skeleton application, with placeholders which may be filled in with the desired behaviour. This removes the need for developers to manually manage the application execution life-cycle (including setup, starting, stopping, recovering from various forms of interruptions), since that is defined and handled once and for all by the framework.

[11] Marcus Fontoura et al. (2000). 'Using domain specific languages to instantiate object-oriented frameworks'. In: *IEE Proceedings–Software* 147.4, pp. 109–116

[12] Inversion of control is sometimes facetiously referred to as the Hollywood Principle: 'don't call us, we'll call you'. In software engineering, this refers to the situation where the life cycle of the application is not the responsibility of the application developer—they just provide various components which are then called as required by the run-time system.

[13] Fayad and D. C. Schmidt 1997

The addition of declarations

Frameworks are found everywhere: in the domain of mobile applications, Web programming, to gaming platforms.[14] Lately, a trend has been emerging, where frameworks make use of domain-specific declarations as input.[15] These declarations are intended to dictate the structure, resource permissions, and behaviour of applications. For example, the `Manifest` file required by Android applications declares which resources of the mobile phone the application is authorised to use.[16] Resources are any potentially sensitive sources or sinks, whether real devices—*e.g.*, camera, microphone—or virtual ones—*e.g.*, address book, the Internet. Such declarations not only allow the framework to better answer emerging challenges such as privacy concerns, but also increase the potential to provide support to the developer, and give a user insight into how their sensitive information is used. This dissertation is focused on these *declaration-driven frameworks* as a way to answer privacy challenges. Increasing user insight does, however, assume that the declarations are presented to the user in a comprehensible format. This is an area on which little work has yet been done.

[14] Mike McShaffry and David Graham (2012). *Game Coding Complete*. 4th edition. Independence, KY, USA: Cengage Learning PTR.

[15] Rick Rogers et al. (2009). *Android Application Development: Programming with the Google SDK*. Beijing, China: O'Reilly.; Snoyman 2012; and Cassou, Bruneau et al. 2012

[16] Rogers et al. 2009

2.3 *Open platforms*

Recently, we are seeing an explosion of new application domains, such as mobile devices, using declaration-driven frameworks to support the open platform model as defined by Balland, et al.[17]

When we refer to open platforms, we mean platforms with (1) public *programming interfaces*, which give access to (2) *shared resources* for applications. They include (3) a *run-time* environment for applications, and contribution of applications is (4) *open* to non-certified, third-party developers. Examples include Android and Apple's iOS, but also the Facebook platform, among many others.[18] Because it is an attractive business model to offer a platform for which third-party developers can easily write applications for end users to install, the open platform model is being widely adopted. When we refer to the *platform owner*, we mean the entity responsible for providing developers with a programming interface and end users with access to an application repository where the third-party developers make their applications available. In the case of Android, the platform owner is Google.

[17] Balland and Consel 2010

[18] Dave Mark and Jeff LaMarche (2009). *Beginning iPhone Development: Exploring the iPhone SDK*. Apress; Feiler 2008; Chrome developers (2015). *Developing Chrome Extensions: Declare Permissions*. `https://developer.chrome.com/extensions/declare_permissions`. Accessed: February 2015; and Matthias Kalle Dalheimer (2010). *Programming with QT: Writing portable GUI applications on Unix and Win32*. O'Reilly Media

Challenges and requirements

These novel application domains pose new challenges. For example, mobile computing platforms expose sensitive shared resources, such as the camera or contact list, to third-party, potentially untrustworthy developers. It has been shown that in Android, routine abuse of these resources is widespread.[19] Among declaration-driven frameworks, we identify a spectrum of approaches to dealing with restrictions of resource usage. Examples range from fully dynamic,

[19] Wei et al. 2012; and Alexandre Bartel et al. (2012). 'Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android'. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, pp. 274–277.

as in Android, to static, as in DiaSuite,[20] an existing declaration-driven approach. Finally, to encourage adoption, platform owners want to facilitate the development process as much as possible.

[20] Cassou, Bruneau et al. 2012

Unfortunately, these stakeholders are not without conflicts of interest. For example, the platform that can offer the most novel or reasonably-priced applications to the user will be most attractive, so platform owners are interested in attracting as many developers as possible. One of the ways to do this is to offer developers a way to earn revenue from the users of their applications. Apart from directly charging a fee to be allowed to download them, what frequently happens is that developers embed advertising into their applications.[21] However, advertising is most lucrative when as much private data as possible can be scraped from the users' devices, to the extent that users can be uniquely identified. This is at odds with the principle of respecting users' privacy, but poses a difficult choice to platform owners.

[21] Stevens et al. 2012

On the one hand, the platform owner could allow developers to gather as much data as possible from users, allowing them to earn as much as possible from advertisers. This way they would stimulate the creation of a rich selection of low-cost applications to the user. On the other hand, they could protect the users' privacy, at the risk of driving away advertisers, and as a result developers. This threatens to leave the ecosystem less attractive to the end user, because of a lack of applications.

When considering declaration-driven frameworks in open platforms, we therefore observe that the different stakeholders have various concerns, summarised here:

- The end user would like clear insight into resource usage by third-party applications: not just *which* resources—*e.g.*, camera, address book—are requested, also *how* they are used—*e.g.*, reading one address book entry and sending it to a friend by SMS.
- The platform owner wants to facilitate the development process as much as possible, to encourage adoption of the platform. Restricting malicious behaviour is also beneficial, since this would increase user confidence in the platform.
- The third-party developer is interested in high-level programming support and abstraction from platform details, leading to greater portability—*e.g.*, hardware-agnostic implementations.

Following from these concerns, we propose more precise requirements for frameworks supporting open platforms. They are not arbitrary: by comparing them to prevalent and successful frameworks, we validate that these are emergent requirements of real-world stakeholders.

[**Req1**: transparency] The user would like clarity on which shared resources will be used. *Resource declarations* should therefore specify the sources and sinks of potentially sensitive data a given application uses, as well as possible side-effects. On mobile computing

platforms, examples include camera or Internet access. This would allow a user to make an informed decision on whether they trust the application enough to execute it.

[**Req2**: containment] The *data reachability* should be constrained to avoid privacy leaks.[22] Potential leaks can be predicted, by determining whether a control flow path exists between the different components constituting an application, which may have access to various sensitive resources. In Android, for example, allowing an application to access both the Internet and photos, implies that photos can potentially be exfiltrated to an arbitrary server. The same applies to Apple's iOS: if the user gives permission to access a given resource, no information is provided regarding what the data will be used for.

[22] Damien Cassou, Emilie Balland et al. (2011). '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. Waikiki, Honolulu, HI, USA: ACM, pp. 431–440.

[**Req3**: support] Tailored *programming support* for the developer can and should be derived from the declarations, since these provide hints towards the desired structure and behaviour of the application. For example, if the declarations do not allow a certain resource to be used, its API need not be available to the developer. This also avoids confusion and clutter during implementation.

[**Req4**: conformance] *Conformance checking*, whether static or dynamic, should be performed between the specifications and the implementation. This way a user can trust the declarations to be meaningful for the application.

These are the criteria according to which we judge our methodology in the following part.

The Sense/Compute/Control model

Our development methodology and toolkit closely follows the Sense/Compute/Control (SCC) architectural style described by Taylor et al.[23] We therefore present the most important concepts here. The next chapter explains how we adapt the SCC model to our work.

[23] Taylor, Medvidovic and Dashofy 2009

The SCC 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 architectural style is also a good fit for open platforms such as those found in the domain of mobile computing.

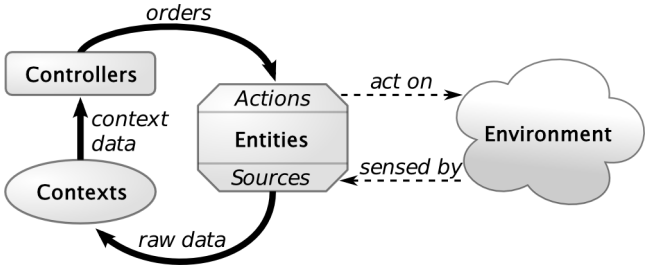


Figure 1: The *Sense/Compute/Control* paradigm. Illustration adapted from Cassou, Bruneau et al. 2012.

As depicted in Figure 1, this architectural pattern consists of three types of components: (1) *entities* correspond to managed[24] resources, whether hardware or virtual, supplying data; (2) *context components* process data; (3) *controller components* use this information to control the environment by triggering actions on entities. Furthermore, all components are reactive. That is, sources are the only components that may decide to publish at any time—*e.g.*, as a result of a changed environmental factor, such as the temperature reported by a sensor increasing in value. Contexts and controllers, on the other hand, may only be activated as a result of a value being published by a component they are subscribed to. This decomposition of applications into processing blocks on the one hand, and control flow on the other, makes data reachability explicit, and isolation more natural, which gives the potential to answer **Req1** and **Req2** effectively. This makes SCC a suitable model for our approach.

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

When targeting a specific domain such as building automation or mobile phones, the platform owner defines a taxonomy of resources for applications in this domain. On mobile devices, for example, this includes the camera, contact list, Internet, *etc.*, whereas in home automation the taxonomy might include temperature and atmospheric pressure sensors, and actuators to open windows or turn on the lighting.[25]

[25] William C. Mann (2005). *Smart Technology for Aging, Disability, and Independence: The State of the Science*. 1st edition. Hoboken, NJ, USA: John Wiley and Sons.

2.4 *Previous work on DiaSuite*

Before the work on this thesis commenced, a number of other projects looked at solving various specific problems by making contributions to the DiaSuite methodology, mostly by adding specific features to the specification language. In the category of extensions to the declaration language, especially the work by Enard[26] and Gatti[27] are good examples. Enard's work added facilities to the design language for specifying how applications should handle exceptions, by means of rules which would be enforced at implementation time. Gatti's work concentrated on Quality of Service (QoS) concerns such as maximum response times of function calls, which are relevant to domains such as aeronautics, where high-assurance software is required. These examples illustrate that rich and expressive declaration languages can be used to enforce guarantees for a wide range of application properties.

[26] Quentin Enard (2013). 'Development of dependable applications: a design-driven approach'. PhD thesis. Université Sciences et Technologies–Bordeaux I.

[27] Stéphanie Gatti (2014). 'A stepwise approach for integrating QoS throughout software development process'. PhD thesis. Université de Bordeaux.

Even more relevant to this work is Cassou's dissertation,[28] which provided much inspiration for the direction taken in the present work. Notably, initial steps were taken to formalise the methodology involving a compiler which produces a tailored programming framework from a specification written by an application architect. This work aims to continue in that spirit, as well as answer the main question posed there, namely: what are the general requirements of a target programming language to be able to host an implemented instance of our methodology?

[28] Cassou 2011

Part II

Presenting the Methodology

3
Software development with tailored frameworks

This part presents the theoretical results of our work. It summarises the definitions that are needed to understand the practical component of this dissertation. It is split into two chapters: this chapter presents the overview of our methodology, and the second, Chapter 4, gives the semantics for the compiler at the heart of our approach.

In Section 3.1, we refine the Sense/Compute/Control model that was introduced previously, by instantiating it with concepts from the mobile computing domain. Concepts from the SCC model such as resources are relevant to our problem domain, and we argue that the Sense/Compute/Control paradigm is a suitable model for mobile applications.

Next, Section 3.2 introduces the Core DiaSpec specification language. It is used as a vehicle for experimentation in the rest of this work. The grammar is given, along with the informal specification for a mobile application, which will be our running example. Here, the relationship between resources, contexts, and controllers is explained, as well as the restrictions on their interaction—*e.g.*, strict reactivity. These restrictions notably include that the application should not be able to leak private user data to unauthorised sinks, such as the Internet.

The development phases associated with creating an application using our methodology are presented in Section 3.3. An overview of the process is given, starting from the specification of the application, through compiling the specification into a tailored programming framework, to implementing the application.

Essential to our methodology is the Core DiaSpec specification language and the accompanying semantics. After having dealt with the overview of the methodology, the semantics are introduced in the next chapter.

3.1 Sense/Compute/Control and mobile computing

Our running example comes from the domain of mobile computing, so we specialise the SCC model, introduced in Chapter 2, in terms of concepts relevant to our problem environment. Figure 2 illustrates the relationship between resources and the application in

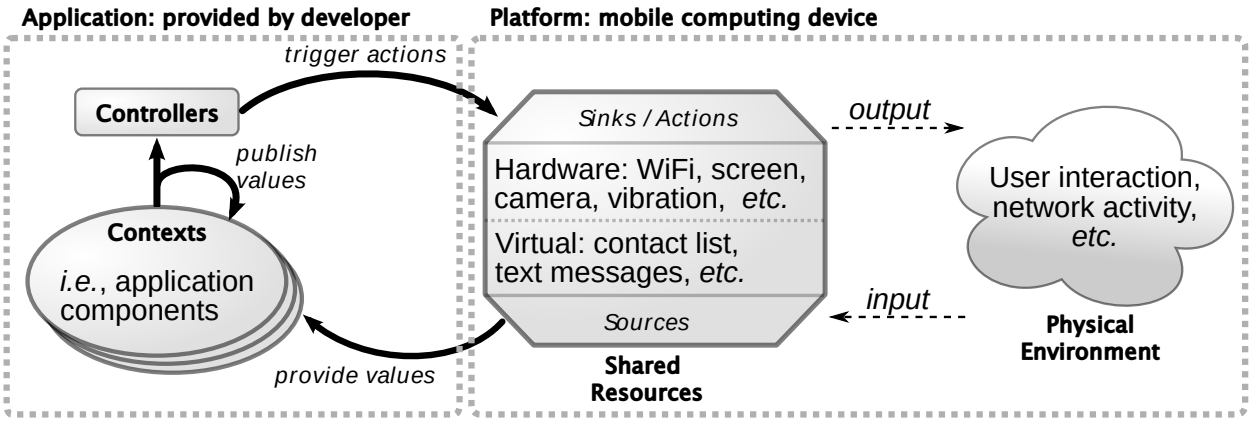


Figure 2: The SCC model as applied to the domain of mobile computing.

more detail, as well as who is responsible for each aspect. The physical environment is where our application gets stimuli from, which includes wireless network activity and interaction by the user, such as pressing buttons on the touchscreen. The application then acts on the environment by printing output to the screen, vibrating, or sending a text message.

Note that both hardware and virtual devices are considered resources in our model: virtual devices represent data that belongs to the user—*e.g.*, contact list or text messages—and hardware devices refer to elements such as the vibration motor, geolocation device (GPS), or screen. The resources thus provided by a mobile computing device, such as the integrated camera or wireless network interface, are shared by all the installed applications. We assume that this is mediated by the operating system: applications are provided with a programming interface (API) to receive input and manipulate the environment. All these sources and sinks, whether hardware or virtual, are presented to the application via a consistent API; that is, the method for accessing one resource or another should not require the application developer to undertake vastly different actions.

Together, the shared resources plus the run-time environment that manages access to them, make up the platform that applications run on. We refer to this as an *open platform*, because third-party developers can provide applications to end users via an application store, which is managed by the platform owner, as defined in Section 2.3.

Crucial to the success of our approach is that the underlying architecture makes a clear separation between components belonging to the platform (the resources), and application-specific components. Therefore, these applications are composed of *contexts* and *controllers* as defined by Taylor, et al.[1] Contexts are the application components that may subscribe to events published by sources—*e.g.*, an incoming call or a change of network interface status. These contexts in turn may publish new values, to which other contexts and controllers may be subscribed. Controllers are granted per-

[1] Taylor, Medvidovic and Dashofy 2009

mission to trigger actions, which is how the application returns information to the user after having performed its task.

Running example of mobile application

We now introduce the application that serves as the running example throughout this dissertation. We base our example on a well-known application, that allows a user to take a picture, which is then instantly processed using a visual filter. For our example, the application should be allowed to show the picture to the user; we want to prevent any other flow, such as sending the picture over the Internet. Since the hypothetical application is distributed free of charge, supported by advertisement revenue, it relies on an advertisement component. Our threat model is that this component tries to exfiltrate the picture to a third-party server.

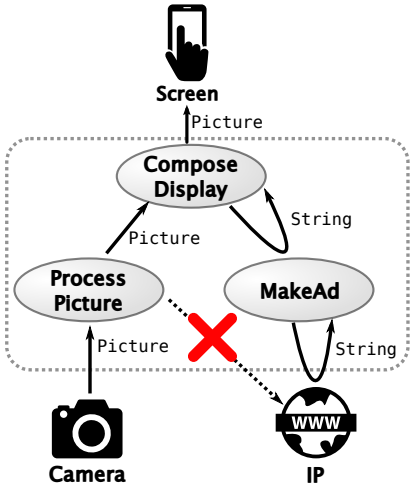


Figure 3: Simplified schematic of the design of the example application. We do not want the picture to be able to leak to the Internet. The application developer provides the components indicated by the grey ovals, the dotted grey square indicates the boundary between the platform and the application.

In terms of the Sense/Compute/Control model, it makes sense to decompose our example application as informally illustrated in Figure 3. Note that the arrows signal subscription relations, and that they are annotated by the type of values that may be transmitted via each channel. The component which applies the visual filter (called **Process Picture** in the diagram) is triggered when the **Camera** device indicates that it has a new picture available. After applying the filter, the **Process Picture** component publishes the new image, to which the **Compose Display** component is subscribed. Before passing the picture along to the hardware device **Screen**, it requests (pulls) the advertisement text from the **MakeAd** component. In our example, **MakeAd** is explicitly given access to the Internet in the specification. Note that when **Compose Display** makes a request to **MakeAd**, our model does not allow for an argument to be passed along with the request. Therefore, the image cannot be communicated to **MakeAd**. In turn, **MakeAd** makes a request to the **IP** resource, again without a parameter. This is a simplification compared to a real-world application, since a specific URL should be queried. Our model supposes that the **IP** device

has already been configured with the correct URL to download the advertisement text. This is a valid simplification: in a real implementation the specification would allow for two types of requests: with and without a parameter. **Compose Display** would make a request *without parameter* to **MakeAd** to prevent the picture leaking, and **MakeAd** would be given permission to make a request to **IP** *with* a parameter, hence being able to download an advertisement from the correct URL. This simplification is made to keep our specification language grammar as uncluttered as possible—it would be easy to add such an extension in practice. Disabling covert channels of communication such as shared memory or SRFI 39 parameter objects[2] is dealt with in the implementations, and are discussed in the following chapters.

[2] Marc Feeley (2003). 'SRFI 39: Parameter objects'. In: *Scheme Requests for Implementation*. Ed. by Arthur A. Gleckler. http://srfi.schemers.org/srfi-39/srfi-39.html. Accessed: September 2015. Published online

This decomposition of the application into components allows us to show that the image cannot be communicated to the Internet, whether intentionally by the application developer, or by using a malicious advertisement provider. 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.

3.2 *Core DiaSpec declaration language*

This section introduces Core DiaSpec, the domain-specific language[3] we define for specifying applications. The grammar of the declaration language is presented in Figure 4, and is adapted from Cassou, Bruneau et al. 2012, keeping only essential constructs. An application specification consists of a list of `Declarations`.

[3] Martin Fowler (2010). *Domain-specific languages*. Pearson Education

```
Specification -> Declaration*
Declaration   -> Resource | Context | Controller
Resource      -> Source | Action

Source   -> source sourceName as Type
Action   -> action actionName as Type

Type     -> Bool | Int | String | Picture | ...

Context            -> context contextName as Type
                      { ContextInteraction }
ContextInteraction -> when ( required GetData?
                           | provided (sourceName | contextName)
                                GetData? PublishSpec)
GetData       -> get (sourceName | contextName)
PublishSpec   -> (always | maybe) publish

Controller    -> controller controllerName
                 { ControllerInteraction }
ControllerInteraction -> when provided contextName
                         do actionName
```

Figure 4: The grammar of Core DiaSpec, our specification language. It will serve as an example in the rest of this work. Keywords are in bold, terminals in italic, and rules in normal font.

Resources. Both hardware and virtual resources (such as camera, GPS, contact list, *etc.*) are defined and implemented by the platform,

and are inherent to the application domain. Sources and actions respectively return or accept values of a fixed type. Context and controller declarations each contain one interaction contract (the `ContextInteraction` and `ControllerInteraction` rules),[4] which prescribe how they interact. Specifically, they declare the conditions under which a component must be activated. This might be the publication of a new value by some other component. They also declare which, if any, data sources they may consult while they are activated. We explain the two types of interaction contracts as follows.

[4] Cassou, Balland et al. 2011

Context interaction contracts. A context can be activated by either (1) another component requesting its value (a **`when required`** interaction contract) or (2) a publication of a value by another component (a **`when provided`** *`component`* interaction contract). When activated, a context component may be allowed to pull data (denoted by the optional `GetData` rule, for example **`get`** `Camera`, meaning that a context may query the camera resource) from a source or another context.

Contexts which may be pulled from, must have a corresponding **`when required`** contract. That is, one may not put an arbitrary context *Y* in the interaction contract **`when provided`** `X` **`get`** `Y`, unless the *Y* context is defined as **`when required`**.

Finally, a publication-activated context might optionally be required to publish when triggered (defined by `PublishSpec`). Note that **`when required`** contexts have no publication specification, since they are only activated by pulling, and hence by definition return their values directly to the component which polled them. If a context is defined to **`maybe publish`**, it has no constraints, but if it must **`always publish`**, an error should be raised if the context does not publish a new value when it is activated.

Controller interaction contracts. When activated, controller components can send orders, using the actuating interfaces of components they have access to (declared using **`do`** *`actionName`*), for example displaying an image to the screen or sending email. As defined in the grammar, controllers may only subscribe to contexts, not directly to sources.

Example specification. Figure 5 shows a possible specification of the example application in Figure 3. The terms between the curly braces are interaction contracts, which determine the subscription relations between components. When the user presses a hardware button, it publishes a new value, which triggers the context that processes the picture, `ProcessPicture`. This context queries the camera, and after applying a filter to the image, `ProcessPicture` must publish (in accordance with **`always publish`**, line 4). As a result, `ComposeDisplay` is activated. Before displaying the picture to the screen, `ComposeDisplay` overlays an advertisement. It is down-

```
context ProcessPicture as Picture {
  when provided Button
  get Camera
  always publish
}

context MakeAd as String {
  when required
  get IP
}

context ComposeDisplay as Picture {
  when provided ProcessPicture
  get MakeAd
  maybe publish
}

controller Display {
  when provided ComposeDisplay
  do Screen
}
```

Figure 5: The Core DiaSpec specification for our example image filter application.

loaded from the Internet by `MakeAd` and returned as a string. Since something might go wrong with the download, `ComposeDisplay` is not obliged to publish, hence the **`maybe publish`**. The application architect could have decided to force `ComposeDisplay` to always publish, even if the advertisement could not be downloaded, but we prefer to provide examples of usage for each feature of the specification language.

Each platform provides its own resource taxonomy. This allows the approach to be generally applicable to any application domain or platform. Therefore, we assume that the declarations for `IP`, `Camera`, and `Screen` are provided by the *platform owner*, since they represent shared resources. Their declaration terms are therefore separate from the specification file of the application.

As mentioned Section 3.1, we assume that the `IP` device does not need a URL parameter to fetch the advertisement. Note that writing the specification does not impose much overhead on the developer, since it is lightweight. It represents approximately the same amount of code as would be needed for the method headers corresponding to each component.

3.3 *Phases of application development*

We now explain the two aspects of application development using our methodology. First, we focus on the development phases, that is, the steps the application developer must go through to develop an application using this methodology. Second, Chapter 4 goes into detail concerning the design of the specification compiler, highlighting its key phases. This is the compiler tasked with producing a tailored programming framework from a set of declarations as defined above, which is at the heart of our approach.

Our approach goes further than that presented in Roberts and Johnson 1996: originally their work gives general guidelines for the development of object-oriented programming frameworks, using what they call a 'pattern language': in fact, it is a series of steps for evolving from three example applications, extracting common code into a library, and finally turning that into a set of classes which can be implemented through extension (in the subclass sense).[5] This dissertation guides the development of families of programming frameworks, where the compiler tailors the framework according to the specification of each application. The framework is tailored to each application through the use of a true domain-specific language.

[5] Don Roberts and Ralph Johnson (1996). 'Evolve frameworks into domain-specific languages'. In: *3rd International Conference on Pattern Languages for Programming*. Monticelli, IL, USA

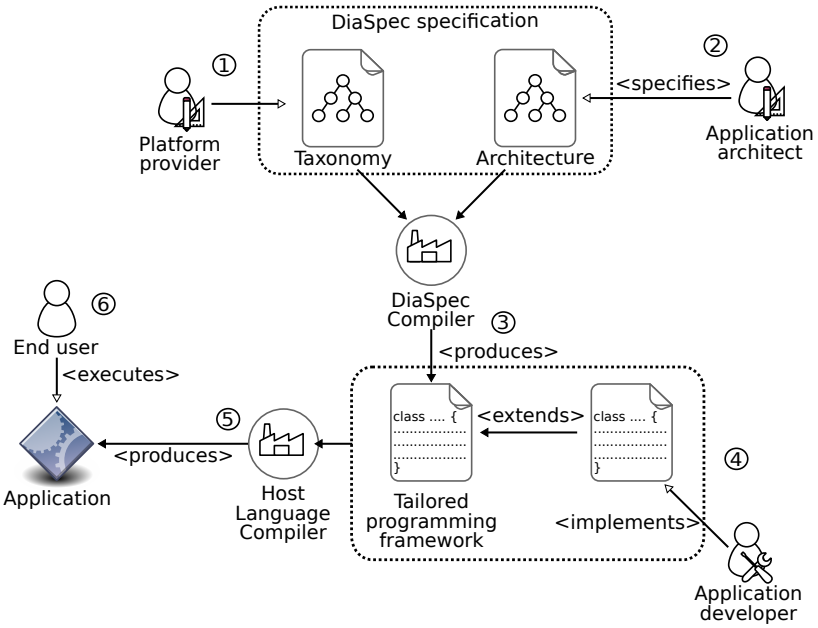


Figure 6: The phases of development, from specification of an application through to execution.

Consider Figure 6. The process of application development starts with ① the platform owner providing a list of available shared resources, called a taxonomy. In the taxonomy, the virtual and hardware devices are enumerated, and given a type. This constitutes the programming interface (API) the application developer uses to access values such as the list of the user's contacts, or to be able to display to the screen. The taxonomy corresponding to the resources used in the example application is displayed in Figure 7. Here we see that the developer has access to the screen, to which an image may be displayed, as well as the camera which can be used to capture images. There are also the `IP` and `Geo` sources, which in our prototype provide access to the Internet and geolocation information. The `Button` device models a hardware button. If pressed by the user, it publishes the value `true`. Note that the developer only uses `Button`, `Screen`, `Camera` and `IP` in Figure 5.

Now that the taxonomy is known, the application architect ② should write the specification of the application, for example the

```
source Button as Boolean

action Screen as Picture

source Camera as Picture

source IP     as String

source Geo    as String
```

Figure 7: The taxonomy of devices for our mobile computing example.

one in Figure 5. The application description together with the taxonomy of resources it uses forms the specification of the application which is to be compiled ③. The specification compiler is at the heart of our methodology, and is to turn the declarations into concrete programming artefacts—*e.g.*, classes, modules, type signatures. The following section details the design of our compiler. The product of this compiler constitutes an application programming framework which the application developer uses to implement the concrete application ④. Note that the application developer and application architect need not be different people, we simply denote the different roles with specific names.

Although application programming frameworks are already a widespread and well-known concept, as discussed in Section 2.2, the output of this compiler is a programming framework *tailored to the specific application*. That is, the framework must provide specific support, guidance and restrictions to the application developer, based on the specification provided in the previous step. The fact that the programming framework is tailored allows a much higher degree of assurance to the end-user that the application does what is advertised, and nothing more, since there is a semantic correspondence between the specification and the application. While this could also be achieved using dynamic checks in the final application, our methodology allows applications to be correct by construction.

Once the developer has written the implementation of the application using the tailored framework, the application code (excluding the generated framework) and the specification should together be sent to the application store, which is managed by the platform owner. There, the specification is recompiled into a framework (ensuring that the developer has not modified the framework in any way), and this framework plus the developer's source code is compiled into an application binary. The resulting object code can be run on the end-user device ⑤. The reason for recompilation by the platform owner for the application store is that the platform owner, whom the end-user presumably trusts more than the third-party application developer, can then verify that the tailored framework has not been tampered with by the developer, for example by removing checks or security features.[6]

This process uses an unmodified compiler for the host language, and produces object code as usual. Finally, this application can

[6] It is interesting to note the recent attack on Chinese Apple iOS application developers. Some developers had unwittingly been using a trojanised version of Xcode, the Apple IDE. This subverted version of Xcode was injecting malicious code into the applications compiled with it, thus infecting end user devices. The injected code manipulated devices to send data to servers controlled by the attackers. If an approach such as ours were used, the recompilation by the platform owner would render such an attack impossible. For details on the attack, see C. Xiao 2015.

be downloaded from the application store and run by the user ⑥. The run-time system library on the end-user's device may also include features to check that the application is not misbehaving with regard to the specification. This is detailed in the following chapters.

Software maintenance and evolution. We emphasise that our methodology does not specifically address the evolution aspect of production software. In our methodology, when the platform is updated, a new declaration compiler needs to be distributed. Compare this with the distribution of an updated programming framework: a developer might need to modify their application to be compatible with an updated version of the programming framework. Our methodology is equivalent: after recompiling the application specification with a new version of the declaration compiler, it may be necessary to adapt the application code to the newly produced framework, depending on the changes the platform owner has made. We argue that our approach is therefore not at a disadvantage compared to the widely accepted situation, namely non-tailored application programming frameworks for mainstream open platforms.

Revisiting the requirements for our development environment

The goal of our approach is to support the developer, and to ensure certain behaviours. We now refine the requirements as identified in Section 2.3, according to our presented phased development approach. We identify the concrete result of the requirements, that will be needed in the framework. Restrictions are for when we want to ensure certain properties—*e.g.*, the program should not be permitted to access private user data. Support is when the developer should be guided—*e.g.*, by being provided with a specialised API. We instantiate each of the requirements for our case study.

[**Req1**: transparency]

- The user should be given the opportunity to approve sensitive operations.
- Restrictions: once the user approves the specification, each component should only have access to the resources explicitly granted. For example, in Figure 5, only the `MakeAd` context should be able to query the `IP` device. Also, `MakeAd` should not have access to any image from the `Camera`.

[**Req2**: containment]

- Restrictions: the developer should not be able to activate components by arbitrarily broadcasting or polling components, except via framework methods. This control flow restriction is how

we enforce data reachability. This is a coarse-grained method, but we use it to avoid doing full static analyses on the code, for example.

[**Req3**: support]

- Support: the publication system should be transparent to the developer. That is, the developer should merely have to write functions that return values to be published, and not have to look up which components are subscribed, *etc.* The framework must take care of the subscription and message delivery steps.
- Support: API calls for accessing resources should be made available as needed, exclusively to the components authorised to use them, based on the declarations. For example, `ComposeDisplay` should have the `MakeAd` API in scope, easily accessible.
- Support: all declared components require an implementation. If any are missing, the developer should be warned.

In summary, the developer should be given informative warnings if the application does not conform to the specifications in any way.

[**Req4**: conformance]

- The application should be checked to conform to the specification. If a component fails to broadcast when promised, tries to initiate unauthorised access to a resource, or otherwise deviates from the specification, the verification should fail.

Conclusion

In conclusion, this section has demonstrated that the SCC paradigm can effectively model applications in the domain of mobile computing. We have introduced the running example application which will be used in the rest of this dissertation to illustrate our methodology. We have presented the minimalistic declaration language, based on the DiaSuite declaration language. Although minimalistic, we have argued that we can express applications using the essential constructs it offers. We have presented the abstract overview of our methodology: a compiler which takes application specifications as input, and produces tailored programming frameworks. Finally, we have refined the requirements given in Section 2.3, which we will use to evaluate the implementations of our methodology presented in the following part of this work.

Concretely, the contributions of this chapter are: (1) showing the applicability of the SCC paradigm and more particularly the DiaSuite methodology to the domain of mobile computing, (2) reflection on the distribution model of applications, which still needs further research, (3) refining the requirements for open platforms

put forth by Balland and Consel 2010 to the domain of mobile computing, and (4) abstracting the concepts and design decisions of DiaSuite from their initial implementation in Java.

Next, we present our translation of these requirements into a declaration compiler which produces an abstract programming framework.

4
Semantics of the declarations

This chapter details the semantics of the declaration compiler described previously. To be able to implement our methodology, we must clearly specify the static semantics of each aspect of the declaration language.

The phases of a declaration compiler designed according to our methodology are listed and explained in Section 4.1. This declaration compiler is the tool that enables our methodology, and produces a *tailored programming framework* which enforces the properties defined by the application architect.

In Section 4.2 we provide typing rules for Core DiaSpec, our specification language. The typing rules are illustrated using the PLT Redex system.[1] The typing rules give criteria for what constitutes a valid and coherent specification. Data reachability is defined formally in terms of reachability on a graph of resources and contexts.

[1] Matthias Felleisen, Robert Bruce Findler and Matthew Flatt (2009). *Semantics Engineering with PLT Redex*. 1st edition. The MIT Press.

After the type checking phase, the compiler must output terms in the host programming language; we therefore discuss the semantics of the declaration language in Section 4.3. This is the key compiler phase which includes novel ideas contributed in this thesis. We provide pseudocode for the implementation of the compiler.

Section 4.4 deals with trade-offs that must be made concerning the phase in which guarantees are to be enforced: the requirements previously outlined can be implemented statically or dynamically without impacting their integrity.

Finally, a discussion of our methodology compared to other mainstream approaches to solve related problems is provided in Section 4.5.

4.1 Phases of the declaration compiler

We begin by exploring the declaration compiler, from step ③ in Figure 6 of the previous chapter. The phases of our compiler are illustrated in Figure 8. As stated previously, the output of this compiler is a programming framework tailored according to the specification of the application.

The phases of this compiler can be divided into (1) bookkeeping and preliminaries, (2) a type checking phase, and (3) the output con-

struction phase, or back-end. Our methodology contains technical contributions in the latter two phases, which are detailed in this section.

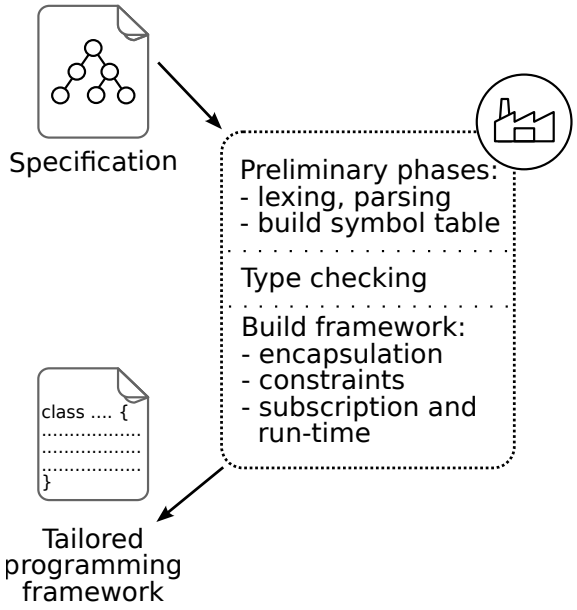


Figure 8: The global phase diagram for the specification compiler. It outputs the framework intended to be used for implementing the application. The type checking and framework construction phases are discussed in detail in this chapter.

Preliminaries. The declaration compiler starts with familiar preliminary phases (the topmost section of the compiler indicated in Figure 8), namely parsing the human-written specifications into an abstract syntax tree (AST). The specification in Figure 5 is an example of a valid input. The compiler also builds the symbol table for the specification from the AST, which includes the declared output type of each component, the activation condition, publication requirement, and possibly whether it has permission to access another component—*i.e.,* using a **`get`** or **`do`** clause.

We consider this phase to be self-explanatory and therefore do not go into more detail. For our purposes it is also not important how the platform owner decides to implement these phases in terms of concrete data structures, algorithms, *etc.* If the application architect has provided a syntactically correct specification, the preliminary phases will complete successfully.

Type checking. The next step is to run a type checker on the AST, which will ensure that the specification is semantically coherent. A set of typing rules for our declaration language is given in Section 4.2.

In informal terms, these rules ensure that all referenced components exist in the symbol table and have been correctly declared. Compatibility of interaction contracts is checked—*e.g.,* a **`get`** `X` clause in some context C_1 implies that X has a **`when required`** activation condition. If not, the specification is incoherent. Finally, after type checking, it is possible to infer the input types of contexts and controllers. For example, if a context C_2 has a clause

when provided Camera, we can infer that C_2 must have input type **Picture**, since the type checking phase will have determined that (1) the component Camera exists, and (2) it has output type **Picture** (declared in Figure 7, line 5). This information is added to the symbol table.

Build output abstract syntax. Once the specifications have been checked for coherence, construction of the output—*i.e.,* the programming framework—can commence. This is the key phase of the compiler on which our methodology relies. Pseudocode for this phase is presented and discussed in Section 4.3. Note that we consider the output described in this chapter to be an abstract framework, in the sense that it represents the structure of the framework. A host language has explicitly not yet been chose. To be usable in a concrete host language, this abstract framework still needs to be translated into an implementation. This process, along with the decisions the framework designer must make, is described in the following chapter.

The abstract framework construction phase consists of building the output abstract syntax, ready to be written to output files or modules that the application developer can use. Depending on the choice of the target language—*i.e.,* the host language for the application—the output might be a collection of classes, macros, or other terms in the target language, but these details are not considered in this chapter.

As shown in Figure 8, the most important aspects that need to be addressed in this phase are as follows.

1. Generate encapsulation per-component. This may be via classes, modules, functions, *etc.*, depending on host language.

2. Construct contracts for the types of values (input/output of contexts and controllers) to handle the constraints on application behaviour. These might be in the form of function types, function contracts, dynamic guards, or assertions in the code, *etc.*

3. Produce generic run-time code which will deliver published messages to subscribed components, common boilerplate code, infrastructure for subscription/publication, *etc.*

There is a spectrum of choices that can be made regarding the implementation of the checks and guards produced in this phase. The checks can be implemented statically or dynamically, which, as we show in the following sections, has profound implications for the behaviour of the system. Furthermore, it is not necessary to choose static or dynamic checks globally, but rather however the platform owner sees fit. The platform owner may even vary the stage in which a particular check is performed per-component. In the following sections we illustrate the implications of such choices. These choices may also engender a non-functional impact:

for example, dynamic checks might increase delays in the final application.

In the context of Figure 6, for each particular check, the platform owner may choose to perform it at stage ⑤ (static) or stage ⑥ (dynamic). Considering systems that are in use in production, we see a wide spectrum of choices made in this regard. In the Discussion section of this chapter, we evaluate the various advantages and disadvantages of each approach (see Section 4.5).

Final output phase and cleanup. Finally, the produced abstract syntax needs to be converted into concrete syntax and perhaps written to files. This process, as well as any cleanup that needs to be done, is considered to be an implementation detail, and is not further dealt with.

4.2 Core DiaSpec: typing rules

This section details the rules that are enforced when type checking a specification. The rules are implemented using PLT Redex,[2] a tool that provides a DSL to specify a grammar and type rules of a language. From that model, one can produce both the presentation format used in this section, and a concrete implementation of a type checker. This type checker is used in the Racket prototype presented in Chapter 6. The type system consist of a set of static judgement rules, presented in the style of Damas and Milner.[3]

[2] Felleisen, Findler and Flatt 2009

[3] Luis Damas and Robin Milner (1982). 'Principal Type-schemes for Functional Programs'. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, pp. 207–212.

Note that the syntax of Core DiaSpec as presented in Figure 4 differs slightly from that which is used in this section: here we use Scheme-style terms enclosed in parentheses. Curly braces from the original grammar are replaced with square brackets surrounding the interaction contracts. However, the terms are equivalent and trivially transcribed. We use this slightly different syntax because it is the one used to construct terms in the PLT Redex tool. Furthermore, note that this section deals with a representative subset of the rules; the full list is included in Appendix A.1.

Extending the grammar. The first step in defining a type system to check declarations in Core DiaSpec is to add type-related terms to the grammar. We extend the grammar provided previously with the production rules displayed in Figure 9. The types of declaration terms are represented by the metavariable t in our model. We also introduce a type environment, Γ. The environment Γ is an association list from variable names (indicated by the metavariable X in the grammar), to types t. The production rule t is necessary to distinguish contexts which may be pulled (CTX-req) from contexts which actively publish values (CTX-prov). The rest of the types are unchanged. The metavariable τ which is provided as an argument to the type terms t, corresponds to the Type rule in the Core DiaSpec grammar previously introduced.

For example, if we started with an empty context, $\Gamma = ()$, we

expect the declaration (**source** S1 **as Bool**) to result in the addition of a pair $(S_1 : (\text{SRC Bool}))$ to the type environment. After that declaration is evaluated the environment should therefore be $\Gamma = ((S_1 : (\text{SRC Bool})))$, a list containing one term. That is, the environment Γ now associates the binding S_1 to a type that denotes a source producing a Boolean. Note that the parentheses are important for the semantics of terms. Reminiscent of Scheme, lists in PLT Redex are represented as a sequence of elements between parentheses, separated by whitespace—*e.g.*, $(e_1\ e_2\ e_3)$.

$$
\begin{aligned}
\Gamma &::= ((X : t)\ ...) \\
t &::= (\text{ACT } \tau) \\
&\quad | \ (\text{SRC } \tau) \\
&\quad | \ (\text{CTX-req } \tau) \\
&\quad | \ (\text{CTX-prov } \tau) \\
&\quad | \ (\text{CTRL})
\end{aligned}
$$

Figure 9: The production rules for the type environment and its terms.

We denote the typing judgements by terms in the form of $\vdash[\![\Gamma, \textit{declaration}, \textit{type}]\!]$. These should be read as logical judgements which hold if and only if there exists a proof tree using the rules provided below. The inputs to $\vdash$ are the current context Γ and the declaration term under consideration; the output *type* is the principal type of the *declaration* term. Procedurally, *type* can be considered the output of a function called $\vdash$, although the rules are simply a collection of valid deductions. If no rule matches a declaration term, the term is considered ill-typed.

Sources and actions. The rules for introducing sources and actions, presented in Figure 10, are simple: as long a declaration does not shadow an existing binding, the new binding may be added to the context. The side condition unique? ensures this property. The judgement unique?$[\![X, \Gamma]\!]$ should be read as returning true if and only if X does not already appear in Γ. For completeness, the Redex definitions of all metafunctions are provided in Appendix A.1.

$$\frac{\text{unique?}[\![X, \Gamma]\!]}{\vdash[\![\Gamma, (\text{source } X \text{ as } \tau), (\text{SRC } \tau)]\!]}\ [\text{intro-src}]$$

$$\frac{\text{unique?}[\![X, \Gamma]\!]}{\vdash[\![\Gamma, (\text{action } X \text{ as } \tau), (\text{ACT } \tau)]\!]}\ [\text{intro-act}]$$

Figure 10: The typing rules for source and action terms. The side condition unique? ensures that bindings are not shadowed in the environment Γ.

As an example, we walk through the intro-src rule. This rule is for introducing sources, and is triggered when a term of the form (**source** X **as** τ) is encountered. We see that there is only one condition above the horizontal line, namely unique?$[\![X, \Gamma]\!]$, which ensures that the variable X is not yet bound in Γ, the environment. Assuming this is the case, we see that the consequence (the part below the line) is that the judgement $\vdash[\![\Gamma, (\text{source } X \text{ as } \tau), (\text{SRC } \tau)]\!]$ holds. This means that given any environment Γ, and the term un-

der consideration, we may deduce that the term has type $(\mathsf{SRC}\ \tau)$. The type τ is the type of the source as declared in Core DiaSpec.

The rule intro-act, for introducing actions, is analogous to the rule for sources, therefore we do not discuss it in detail.

Contexts. The rules for declaring contexts are more intricate. Consider Figure 11, which contains a few representative examples of the full set of typing rules for contexts.

$$\frac{\mathsf{unique?}[\![X, \Gamma]\!]}{\vdash [\![\Gamma, (\mathsf{context}\ X\ \mathsf{as}\ \tau\ [\mathsf{when\ required\ (get\ nothing)}]), (\mathsf{CTX\text{-}req}\ \tau)]\!]}\ \text{[ctx-req-get-ø]}$$

$$\frac{\begin{array}{c}(\mathsf{SRC}\ \tau_2) = \mathsf{lookup}[\![\Gamma, X_2]\!] \\ \mathsf{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\mathsf{context}\ X_1\ \mathsf{as}\ \tau_1\ [\mathsf{when\ required\ (get}\ X_2)]), (\mathsf{CTX\text{-}req}\ \tau_1)]\!]}\ \text{[ctx-req-get-src]}$$

$$\frac{\begin{array}{c}(\mathsf{SRC}\ \tau_2) = \mathsf{lookup}[\![\Gamma, X_2]\!] \\ (\mathsf{CTX\text{-}req}\ \tau_3) = \mathsf{lookup}[\![\Gamma, X_3]\!] \\ \mathsf{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\mathsf{context}\ X_1\ \mathsf{as}\ \tau_1\ [\mathsf{when\ provided}\ X_2\ (\mathsf{get}\ X_3)\ _]), (\mathsf{CTX\text{-}prov}\ \tau_1)]\!]}\ \text{[ctx-onSrc-get-ctx]}$$

$$\frac{\begin{array}{c}(\mathsf{CTX\text{-}prov}\ \tau_2) = \mathsf{lookup}[\![\Gamma, X_2]\!] \\ (\mathsf{CTX\text{-}req}\ \tau_3) = \mathsf{lookup}[\![\Gamma, X_3]\!] \\ \mathsf{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\mathsf{context}\ X_1\ \mathsf{as}\ \tau_1\ [\mathsf{when\ provided}\ X_2\ (\mathsf{get}\ X_3)\ _]), (\mathsf{CTX\text{-}prov}\ \tau_1)]\!]}\ \text{[ctx-onCtx-get-ctx]}$$

Figure 11: A few representative examples of typing rules for context declarations.

There are type rules for each possible combination of activation condition and data requirement. For brevity we only present an extract of the list of rules. Consider the rule `ctx-req-get-ø`, the first rule in Figure 11. It corresponds to a declaration term `(context X as τ [when required (get nothing)])`. This declaration corresponds to the case where a context X is activated because another context executed a **`get`** on it. The context has no data requirement (in accordance with the `(get nothing)` specification), and should return a term of type τ. We see that the judgement `ctx-req-get-ø` associates the type (CTX-req τ) to such a declaration. This corresponds to the fact that it is a context returning values of type τ and activated through being pulled.

The second rule, `ctx-req-get-src`, contains a new premise, involving the lookup metafunction. This rule covers the case where a context X_1 is activated by pull—as before this results in the declaration being judged to have the type (CTX-req τ_1). However, now it also has access to a data requirement X_2 from the `(get X2)` specification. This data requirement results in a new condition,

(SRC τ_2) = lookup$[\![X_2, \Gamma]\!]$. This condition ensures that X_2 exists in the environment Γ and that it is a source which returns a value of type τ_2. In practice the specific value of metavariable τ_2 is irrelevant. If this condition does not hold, the rule does not apply. There is a corresponding rule for the case where the data requirement X_2 is a context with a **`when required`** activation contract (and thus type CTX-req), but it is omitted here.

The third rule, `ctx-onSrc-get-ctx`, is structurally very similar to the preceding rule. This rule differs because it applies to a declaration of a context X_1 which is activated by a subscription to X_2, instead of by pull. The underscore in these rules is at the position of the publication specification, indicating that these rules are valid for any value at that position. The publication requirement only plays a role in the phase after type checking . The first condition, (SRC τ_2) = lookup $[\![\Gamma, X_2]\!]$, stipulates that the component X_2 that activates the context must be a source. The corresponding rule that covers the case where X_2 is a publishing context is the last rule, `ctx-onCtx-get-ctx`. As before, in both cases, the context X_1 also has a data requirement X_3, which in contrast to the second rule is required to be a context, by the condition (CTX-req τ_3) = lookup$[\![\Gamma, X_3]\!]$.

The rules concerning contexts that are not shown here cover the other possible combinations: activation by requirement, or by publication of a context or a source, and finally requiring a context or a source, or requiring nothing. The full list is included in Appendix A.1.

Controllers. Finally, in Figure 12, we present the single rule for introducing controllers.

$$\frac{\begin{array}{c}(\text{CTX-prov } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \\ (\text{ACT } \tau_3) = \text{lookup}[\![\Gamma, X_3]\!] \\ \text{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\text{controller } X_1 \text{ [when provided } X_2 \text{ do } X_3\text{]}), (\text{CTRL})]\!]}\ \text{[intro-controller]}$$

Figure 12: The typing rule for controller declarations.

In the rule `intro-controller`, we see that it may only be activated by a context X_2 with type (CTX-prov τ_2), as opposed to a source, and that it must reference an action X_3. If these criteria are met, the component is assigned the type `(CTRL)`. The `CTRL` type has no parameter, since controllers do not return any value but instead only trigger actions.

Lists of declarations. Given all the rules for declaring individual components, a full specification can be checked for coherence. A specification is simply an ordered list of declarations. We evaluate the list of declarations from beginning to end, and require compon-

ents to be declared before they are referenced. This is an artificial restriction imposed to keep the semantic model as simple as possible. As a result, feedback loops are statically prevented. Feedback loops are where some component C publishes a new value, and as a result gets reactivated again, causing an *ad infinitum* cycle. In Theorem 1, this is proven to be impossible.

In Figure 13 we present the rules which check a full specification, which is a list of declarations.

$$\frac{\vdash [\![\Gamma, \mathit{declaration}, t]\!] \quad X_{new} = \mathsf{varname}[\![\mathit{declaration}]\!]}{\mathsf{decl\text{-}ok}[\![\Gamma, \mathit{declaration}, (X_{new} : t)]\!]}\ [\text{decl-ok}]$$

$$\frac{}{\mathsf{check\text{-}spec}[\![\Gamma_1, (), \Gamma_1]\!]}\ [\text{empty-spec}]$$

$$\frac{\begin{array}{c}(\mathit{declaration}_1\ \mathit{declaration}_2\ ...) = \mathit{specification} \\ \mathsf{decl\text{-}ok}[\![\Gamma, \mathit{declaration}_1, (X : t)]\!] \\ \mathsf{check\text{-}spec}[\![\mathsf{extend}[\![\Gamma, (X : t)]\!], (\mathit{declaration}_2\ ...), \Gamma_2]\!]\end{array}}{\mathsf{check\text{-}spec}[\![\Gamma, \mathit{specification}, \Gamma_2]\!]}\ [\text{check-spec}]$$

Figure 13: The typing judgement check-spec checks a specification, which is a list of declarations, by tail recursion.

First we present the helper judgement decl-ok, introduced to increase readability of the $\vdash$ rules. It gives the deduction rule that presuming a *declaration*, which introduces binding X_{new}, has type t in an environment Γ, we may deduce that the statement $\mathsf{decl\text{-}ok}[\![\Gamma, \mathit{declaration}, (X_{new} : t)]\!]$ holds. This helper merely formats the output of the $\vdash$ judgement as an element of Γ. That is, a pair associating a binder X_{new} with a type t.

Now, the judgement form check-spec can be used to judge whether a complete specification is well-typed. The `empty-spec` rule declares that $()$, the empty specification, is valid under any type environment Γ_1.

The rule check-spec sequentially checks a list of declarations. The first element $\mathit{declaration}_1$ of the list $\mathit{specification}$ is retrieved through pattern matching. The next condition is $\mathsf{decl\text{-}ok}[\![\Gamma, \mathit{declaration}_1, (X : t)]\!]$, using the decl-ok helper to retrieve the name X and type t belonging to the declaration. These are added to the environment using the metafunction extend, and the rest of the specifications (the list $(\mathit{declaration}_2\ ...)$) is type checked under the new extended environment with a tail-recursive call to the check-spec judgement.

The full list of type rules along with the exact definition of all the metafunctions can be found in Appendix A.1.

Properties

This type system induces a number of desirable properties for a given specification. We give a sketch of the proof that the specification is a directed acyclic dependency graph, where the nodes are

the entities and resources, and the edges represent subscription and action relations.

Theorem 1. *Dependency cycles are impossible, given the above type system. That is, it is impossible for a context C_i to be reactivated (directly or indirectly) as a result of a value it publishes.*

Proof. For some context C_1, it is impossible to subscribe to itself (the base case). One would have to declare `(context C1 .. [when provided C1 ..])`, which is not allowed, since

- C_1 does not yet exist in Γ, hence $\mathsf{lookup}[\![\Gamma, C_1]\!]$ will fail in the `intro-ctx-onCtx-...` rule, or else
- C_1 does exist in Γ, in which case $\mathsf{unique?}[\![C_1, \Gamma]\!]$ will fail.

Similarly, for the inductive case, when we assume C_1 activates a chain of contexts $C_2 ... C_n$, and C_n activates C_1, note that C_n could not yet have been defined when C_1 was defined, hence leading to a contradiction. □

Reachability

One of the requirements mentioned in Section 2.3 is that private data be contained. To determine data is contained in a given specification, we must study the reachability between private sources and public sinks on this graph. First, we define the translation between specifications and graphs algorithmically, in Definition 1.

Definition 1. *Calculating a reachability graph from a specification.*

Given a specification D, well-typed according to $\mathsf{check\text{-}spec}[\![(), D, \Gamma]\!]$, let the reachability graph $G = (V, E)$ with nodes V and edges E.

Let $V = \Gamma$. That is, all sources, actions, contexts and controllers referenced in D are the nodes of G. Construct E as follows:

for each declaration $d \in D$ **do**
  if $d =$ `(source ...)` **then** skip.
  else if $d =$ `(action ...)` **then** skip.
  else if $d =$ `(context C ... [when provided F get H])` **then**
    Insert $(F \to C)$ into E
    Insert $(H \to C)$ into E, unless $H =$ **`nothing`**
  else if $d =$ `(context C ... [when required get H])` **then**
    Insert $(H \to C)$ into E, unless $H =$ **`nothing`**
  else if $d =$ `(controller C [when provided F do H])` **then**
    Insert $(F \to C)$ into E
    Insert $(C \to H)$ into E
  end if
end for

The resulting directed graph G now indicates which actions (sinks) may receive sensitive data from which sources. Graph G is acyclic in accordance with Theorem 1.

Taking the graph thus defined, we may calculate the transitive closure of all sources, and prune those paths not ending at sinks. This will give us a set of arrows of the form *source* $\rightarrow$ *sink*, which can be presented to the user. For example, an arrow such as `Camera` $\rightarrow$ `Screen` might be acceptable to the user, while for privacy reasons `Geo` $\rightarrow$ `IP` might not.

This is clearly in contrast with the Android approach, where the user is only presented with a list of permissions—*e.g.*, GPS and internet—but has no insight into the potential data flow paths, meaning that data from any source may be transmitted to any sink. We therefore conclude that our style of specification allows much more precise insight for the end user into the potential behaviour of an application than the Android platform.

4.3 *The framework construction phase of the compiler*

This section details the translation from declaration terms into abstract programming concepts. For each well-typed case in the grammar, we detail the programming language constructs which should arise. This phase depends on the type checking phase to be able to construct appropriate guards between the components. Note that the terms evaluate to pseudo code; the exact implementation choices determine the precise behaviour and properties of the system. These choices are explored in the next section. We provide the translation rules in the denotational semantics style of Schmidt.[4]

We use the notation $[\![\mathit{declaration}]\!]_{\mathit{eval}} \rightsquigarrow \mathit{term}$ to indicate that a given *declaration* evaluates to the pseudocode *term* on the right-hand side. The terms contain holes, marked by the symbol $\{\ \}_?$, to indicate where a developer is expected to provide code snippets. We use the symbol `NULL` to refer to the unit type, which has only one inhabitant—*e.g.*, `void` in Java, $()$ in Haskell, `NULL` in Common Lisp, which is the type of `NIL`, or simply a singleton set in mathematics.[5]

[4] David A. Schmidt (1986). *Denotational Semantics: A Methodology for Language Development*. Dubuque, IA, USA: William C. Brown Publishers.

[5] Benjamin Pierce (2002). *Types and programming languages*. Cambridge, MA, USA: MIT Press.

Resources

Sources and actions are provided and implemented by the platform owner, and are therefore trusted as an interface to actual hardware or sensitive data. The most important constraint imposed on them is the type of the values provided or accepted, respectively.

$$[\![\texttt{(source } X \texttt{ as } \tau\texttt{)}]\!]_{\mathit{eval}} \quad \rightsquigarrow \quad (\lambda() \quad \{\ \}_? \quad) :: [\![\tau]\!]_{\mathit{type}}$$

The pseudocode for source declarations should be read as follows: the declaration `(source X as τ)` results in a placeholder function for which the developer should provide code. The snippet should not require function arguments (hence the empty parenthesis after the λ), and should return a value in the host language corresponding to the abstract type τ in the specification. The ex-

act translation given by $[\![_]\!]_{type}$ will depend on the chosen target language. For example, the type **`Int`** in the Core DiaSpec specification might translate to `Integer` in Java or `int` in C. The mapping defined by $[\![_]\!]_{type}$ in fact embodies a decision the framework designer must make, comprising a design choice regarding how typing is handled. For example, the framework designer might decide to simply erase these types, which could result in a dynamic and untyped implementation.

$$[\![\texttt{(}\textbf{\texttt{action}}\ X\ \textbf{\texttt{as}}\ \tau\texttt{)}]\!]_{eval} \quad \rightsquigarrow \quad (\lambda(x_2 :: [\![\tau]\!]_{type}) \quad \{\ \}_? \quad)\ ::\ \texttt{NULL}$$

The translation for action resources is similar to sources, except here the snippet provided by the developer should accept one argument, for example an image in the case of `Screen`, and need not return any value. The snippet is expected to have as a side-effect the action expected of it, embodying the influence of the action on the environment. The effectful nature of resources is not encoded in the equations presented here, but is left as an implementation detail for the framework implementer.

Contexts

Introduction of contexts should produce similar program snippets with placeholders. However, contexts are expected to be pure, that is, free of side-effects. Where a resource could reasonably be expected to modify or query the environment, the output from a context should depend only on the inputs it is provided. The exact mechanism used to ensure purity will depend on the host language, and is therefore not reflected in the equations here.

$$[\![\texttt{(}\textbf{\texttt{context}}\ X\ \textbf{\texttt{as}}\ \tau\ \texttt{[}\textbf{\texttt{when required}}\ get\texttt{])}]\!]_{eval} \rightsquigarrow$$
$$(\lambda(x :: [\![get]\!]_{get}) \quad \{\ \}_? \quad)\ ::\ [\![\tau]\!]_{type}$$

Here, the value of the $[\![_]\!]_{get}$ function depends on the access this component should have to a data requirement, if any. If a component has no data requirement (*i.e.*, **`get nothing`**) then the argument x, above, will be unit (*i.e.*, `NULL`, the type with only one inhabitant, which therefore cannot hold any information). On the other hand, if a context may query another component (*i.e.*, **`get`** Y) on activation, the argument x above will be a proxy giving access to that component. That proxy is expected to give authorised access only, and return a value of the same type as Y is declared to provide. Authorised access refers to the fact that only contexts which are explicitly granted access via the specification should be able to access data requirements. Additionally, they may not access other components at arbitrary times, but only after the framework has activated the context, and before it has returned. Outside this time interval, access to any other component should be disallowed.

$$[\![\texttt{(get nothing)}]\!]_{get} \quad \rightsquigarrow \quad \texttt{NULL}$$
$$[\![\texttt{(get } Y\texttt{)}]\!]_{get} \quad \rightsquigarrow \quad (\texttt{NULL} \to [\![Y]\!]_{type})$$

The case for contexts activated by the publication of a value is as follows.

$$[\![\texttt{(context } X \texttt{ as } \tau \texttt{ [when provided } X_2 \; get \; pub\texttt{])}]\!]_{eval} \rightsquigarrow$$
$$(\lambda(x_2 :: [\![X_2]\!]_{type},\ x_3 :: [\![get]\!]_{get}) \quad \{\ \}_? \quad) :: [\![pub, \tau]\!]_{pub}$$

Here, we see the parameter x_2 added. Since this context is activated as a result of X_2 publishing a value, the first argument to the snippet has the type of X_2. The argument x_3 is, as above, the data requirement. This context also has a publication specification *pub*, which may be either **always publish** or **maybe publish**. This option determines the expected output type of the snippet. The exact definition of $[\![_]\!]_{pub}$ will depend on the target language, but the idea is as follows.

$$[\![\textbf{always publish}, \ \tau]\!]_{pub} \quad \rightsquigarrow \quad [\![\tau]\!]_{type}$$
$$[\![\textbf{maybe publish}, \ \tau]\!]_{pub} \quad \rightsquigarrow \quad [\![\tau]\!]_{type} \cup \texttt{NULL}$$

If a component should always publish, the output type will simply be τ as declared in the specification. If the publication is optional, the τ should be wrapped in an option type. The exact implementation of the option type will vary per concrete host language.

Controllers

The translation rule for controllers is very similar to those presented above, except that the proxy term is slightly different, as a result of an action being made available, as opposed to a source.

$$[\![\texttt{(controller } X \texttt{ [when provided } X_2 \texttt{ do } X_3\texttt{])}]\!]_{eval} \rightsquigarrow$$
$$\Big(\lambda\ (x_2 :: [\![X_2]\!]_{type},\ x_3 :: ([\![X_3]\!]_{type} \to \texttt{NULL})) \quad \{\ \}_? \quad \Big) :: \texttt{NULL}$$

Note that now, the argument x_3 is a closure which takes as an argument a value of the type expected by the action X_3. The snippet that the developer should provide may therefore act using this closure, and is expected to return unit.

Tying it all together

With all the translation rules for individual components in place, what remains is to organise communication between components (the subscription relations) and the separation of components (encapsulation). Encapsulation of components entails that two

components C_1 and C_2 should not be able to communicate, except when this is explicitly declared in the specification. Typically the developer's implementations of C_1 and C_2 will therefore need to be in separate namespaces, modules, or some other container, depending on the target language. This is dealt with in detail in the following chapter, which illustrates different concrete implementation techniques.

The delivery of messages is not dealt with explicitly in this chapter, since it amounts to collecting pointers to the declared components $C_1 ... C_n$, launching them on application start-up, and notifying subscribed components when values are published. This can be accomplished by repeatedly iterating over a list of resources, or in a more sophisticated fashion, but the exact implementation is out of the scope of this general outline.

Thus, the output of this phase is an abstract framework. The compiler presented in pseudo code needs to be instantiated for the chosen target language, at which point it is necessary to decide how to encode concepts such as encapsulation into concrete programming language constructs. In conclusion, this section provides a guide for implementing a declaration-driven framework using our methodology. Certain design choices still need to be made, such as the $[\![_]\!]_{type}$ function previously mentioned, but the specifics thereof depend on the choice of host language. These choices will be considered in more detail in the Implementation part, Part III. Another choice the framework designer must make is the execution model. Examples include a model where sources are preemptively polled (active), or where the implementation relies on streams (reactive programming).[6] These are both valid implementation choices, but do not influence the applicability of this methodology.

[6] Antony Courtney (2001). 'Frappé: Functional Reactive Programming in Java'. In: *Practical Aspects of Declarative Languages*. Ed. by I. V. Ramakrishnan. Vol. 1990. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 29–44.; and Gregory H. Cooper and Shriram Krishnamurthi (2004). *FrTime: Functional reactive programming in PLT Scheme*. Tech. rep. CS-03-20. Providence, Rhode Island, USA: Brown University

4.4 *Static* vs. *dynamic checks*

The preceding section details a procedure for implementing a framework compiler, leaving the choice of host language up to the framework designer. For any target language, however, the framework designer must make choices regarding the phase in which to implement checks described in this chapter. This section summarises those checks, which are illustrated in the concrete implementations presented in the remainder of this dissertation. The following part of this dissertation is concerned with the implementation options available for these checks, and goes into detail regarding the impact of choosing a static *vs.* dynamic approach for each one.

We identify the following checks in the abstract framework (that is, before having chosen a target language) presented in this chapter, and remark that each of them can be performed either statically or dynamically.

- Type-check all components to match their declarations. This corresponds to implementing the type judgements provided in

Section 4.3.

- Check that all declared components are implemented. That is, verify that a developer's implementation of an application provides a component for every declared entity.
- Verify that publication requirements are respected.
- Check that any access to resources, such as file system, camera, *etc.*, is always authorised. For example, when the framework activates a context, the context should have access to the data requirements listed in the specification. Outside of this time window, however, the context should be unable to access any other entities.

For each of these checks, we discuss their implementation statically and dynamically, and explore the impact of the choice. We refer to Part III and its discussion in Chapter 7 for these details. We see that these checks can be performed in any order, either statically or dynamically. However, this choice does influence the accuracy of privacy warnings, as well as the ease of use of the framework by the application developer.

Finally, we close this chapter with a comparison between what a framework can offer if implemented as described before, and mainstream declaration-driven frameworks.

4.5 Discussion

Our work asks a different question than has been posed before: we attempt to take a step back and design declaration-driven frameworks from the ground up, where they have usually been retrospective engineering solutions to specific problems such as containment of sensitive data. Our methodology proposes techniques which are applicable in arbitrary languages, as long as they have at least one pre-runtime phase. It is using this pre-runtime phase that we implement the declaration compiler as outlined above.

There have been numerous other approaches to this problem, although as seen in Chapter 2, they are mostly categorised as those based on static program analysis techniques versus those with runtime validations (including platforms like Android and Apple's iOS).

We show that for each requirement as presented in Section 3.3, declaration languages vary widely in expressiveness and precision. Referring to Table 1, we discuss the most commonly used mobile application frameworks, compared to the approach proposed in this dissertation.

Granularity of declarations. At one end of the spectrum, the least expressive declaration language might cover only resource usage (**Req1** as defined in Section 2.3). For example, Facebook and Chrome plugins only require an application developer to specify

Property	Android	Apple's iOS	Core DiaSpec
Transparency (**Req1**)			
Permission display	at install time (stat.)	on 1st access (dyn.)	at install time (stat.)
Granularity (**Req2**)			
Permission scope	Per-app	Per-app, per-resource	Per-component, per-resource
Developer support (**Req3**)			
API	✗ (full API available)	✗ (full API available)	tailored API
Missing components	✓ compiler warning	✗	✓ compile fails
Enforcing (**Req4**)			
Type-check components	n/a	n/a	static or dynamic
Resource access control	dynamic	dynamic (only 1st time)	static (dynamic also possible)

Table 1: Comparison of mobile platform frameworks according to the requirements defined earlier.

the resources required—*e.g.*, cross-site requests, the 'friend list', or the user's birth date.[7] Android declarations go further, enforcing a certain architectural style consisting of different views, called activities, and untyped communication channels between them, called intents.[8] The interactions between these components are controlled by the framework—*i.e.*, the system handles delivery of intents by calling the appropriate activities. The underlying declaration language, used in the `Manifest` files, forces the developer to declare the components and permissions of the application. Having resource declarations in combination with structural declarations potentially allows more insight into what may happen with sensitive information. However, the Android declarations are not expressive enough, since permissions apply to entire applications, not components. Based only on the declarations, a misbehaving application is indistinguishable from a reasonable one. For example, if we know an application may access the Internet and access photos, we do not know what it will send where. On the other hand, if declarations are fine-grained, per-component—*e.g.*, Internet access only allowed for certain views—a user might see that an application cannot exfiltrate sensitive data in the background, if separate components have disjoint permission sets.

[7] Feiler 2008; and Chrome developers 2015

[8] Rogers et al. 2009

On the other end of the expressiveness spectrum are approaches such as DiaSuite.[9] Like Android, the DiaSuite declaration language imposes an architectural style, Sense/Compute/Control. Contrary to Android, resource usage in DiaSuite is part of the architecture and specified at the component level, not globally (**Req2**). The declarations also include constructs dedicated to the interactions—*i.e.*, subscription relations—between the components.[10] This combination allows the developer to declare how components may interact with each other, and which permissions each one has. This kind of control flow restriction is essential to our approach for preventing data leaks.

[9] Cassou, Bruneau et al. 2012

[10] Cassou, Balland et al. 2011

Supporting the application developer. Another difference is that Android does not offer application-tailored programming support. Our approach provides a customised framework, generated from the declarations; the application developer is not meant to modify this generated code. In doing so, APIs for disallowed resources are made unavailable to the developer. This strategy lowers development effort, since only essential API calls are available (**Req3**). By contrast, Android always exposes the entire system API, without regard to the declarations.

Static or dynamic permission declarations. Apple's iOS offers yet another model for resource usage restriction. iOS does not use declarations, but simply prompts the user if and when a sensitive resource such as geolocation is about to be accessed for the first time. The checks are therefore dynamic, and these 'declarations', if one may call them that, the moment a resource is queried. This potentially gives the user the advantage of a high degree of insight into which resources an application uses at what moment, but the current implementation falls short. Once a permission is granted, the application may access the sensitive resource as and when it wishes. Unsurprisingly, a common tactic among malicious applications is to wait for the user to grant a benign request, and subsequently exfiltrate data without raising suspicion.[11]

[11] Yuvraj Agarwal and Malcolm Hall (2013). 'ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing'. In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '13. Taipei, Taiwan: ACM, pp. 97–110.

Android, iOS and DiaSuite therefore all offer resource permission management, but their differing implementation choices influence their efficacy and usability. Android and DiaSuite verify resource usage according to declarations (**Req4**), but iOS only has *a posteriori* declarations. The dynamic checks offered by Android and iOS mean that if a developer tries to access a forbidden resource, an exception is raised. This approach risks aborting the application as a result of uncaught exceptions. This might only be discovered via testing, or worse, by end-users. Tailored programming support is also unavailable. By contrast, in DiaSuite, resource usage is enforced statically. The developer and the end-user can therefore be sure, at compile time, that all permissions required have been granted accordingly.

Conclusion

Having presented our methodology in abstract form, and with the evaluation criteria clarified as in Section 4.5, we are ready to present the two prototype declaration compilers in the following part. We construct the framework generators as explained in Section 4.3. The following part is concluded with an evaluation of the prototypes compared to the requirements stated here.

Part III

Implementations

5
Instantiation of the methodology in Java

In this part of the dissertation, we instantiate the methodology previously presented, developing a declaration compiler for tailored frameworks to support the DiaSuite open platform example. Declaration compiler prototypes are provided targeting two significantly different programming paradigms, namely typed object-oriented and dynamic functional programming. The prototype compilers are available for download,[1] including full source code and an example application. We encourage the reader to experiment with the compilers, and try writing their own specification and implementing an associated application. This chapter is intended to serve the dual purpose of explaining the design choices made during framework implementation, as well as instructing a framework implementer on how to apply our methodology to their host language. Finally, it can also be used as a guide for a developer wishing to implement an application by extending a tailored framework as described in this dissertation.

[1] All our code is available online at `http://people.bordeaux.inria.fr/pwalt/code/frameworks.tgz` and `https://github.com/toothbrush/diaspec`. The download includes a prototype Android application illustrating data theft.

We use Java for the object-oriented, statically typed language, and Racket[2] for the dynamic functional prototype. We hypothesise that the significant contrast between these languages should substantially differentiate the implementations and their non-functional properties—*i.e.,* which programming techniques are used to address the requirements.

[2] Matthew Flatt and PLT (2010). *Reference: Racket*. Tech. rep. PLT-TR-2010-1. `http://racket-lang.org/tr1/`, Version 6.2.1. PLT Design Inc.

In this chapter, we present the Java declaration compiler prototype and example application; we present the Racket prototype in Chapter 6. Each chapter concludes with a discussion highlighting the conformance of the prototypes to the requirements presented in Section 2.3.

Both implementation chapters are structured as follows: a *general description* of the design of the prototype framework in Section 5.1, the *translation* of the declarations into programming language constructs in Section 5.2, an *example implementation* of a context in Section 5.3, and finally an *evaluation* of the prototype with respect to the requirements in Section 5.4.

5.1 Overview of the implementation

The instantiation of our methodology in Java consists of two main conceptual components, namely the compiler, which takes a specification as input, and the set of possible tailored programming frameworks it outputs. These frameworks are a collection of class files, the basic modular unit of code in Java. The compiler itself (corresponding to ③ in Figure 6) is implemented using attribute grammars[3] with the Utrecht University Attribute Grammar Compiler.[4] The UUAGC is an attribute grammar compiler which produces Haskell code, making it easy to write catamorphisms: functions that do to any datatype what *foldr* does to lists. One defines tree walks using the intuitive concepts of inherited and synthesised attributes, while keeping the full expressive power of Haskell. The specification compiler loads the declarations into an abstract syntax tree, which is then processed using an Attribute Grammar to output concrete programming artefacts for each declaration. A thorough discussion of the UUAGC system is out of the scope of this work, therefore we discuss the compiler in general terms. Interested readers are invited to download the attribute grammar code for inspection.[5]

This chapter focuses on the architecture of the tailored programming framework that is produced by the compiler. We also describe the specific techniques used to provide the guarantees identified in Section 4.4.

[3] Pierre Deransart, Martin Jourdan and Bernard Lorho (1988). *Attribute grammars: definitions, systems, and bibliography*. Lecture notes in computer science. Berlin, New York: Springer-Verlag.

[4] S. Doaitse Swierstra, Pablo R. Azero Alcocer and João Saraiva (1999). 'Designing and Implementing Combinator Languages'. In: *Advanced Functional Programming*. Ed. by S. Doaitse Swierstra, José N. Oliveira and Pedro R. Henriques. Vol. 1608. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 150–206.

[5] See `https://github.com/toothbrush/diaspec`.

General design of the tailored framework

The Java prototype generates a tailored framework from the specifications, and is adapted from the system proposed by Cassou, et al.[6] We present the core features here for ease of comparison. Each component declaration is translated into an abstract class, which the developer must extend with an implementation class, as illustrated in Figure 14. We employ usual unified markup language (UML)[7] syntax for the classes on the right of the illustration. This approach is akin to what some refer to as the Template Method behavioural design pattern.[8]

In Figure 14, we generate an abstract method with a type derived from the interaction contract[9] of the component (the blue box, top right), which the developer will have to implement (the green box, bottom right). Access to resources is only given via specialised arguments which are passed to the generated abstract methods. The declarations are implemented as an external domain-specific language (pictured top left) from which a separate compiler produces the framework. The developer must extend this framework (and may regenerate it if the specifications change), but does not modify the generated code directly. The developer's implementation remains separate from framework code. Also note that the generative approach does not prevent reuse of code across applications. For

[6] Cassou, Bruneau et al. 2012

[7] Martin Fowler (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional

[8] Erich Gamma et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st edition. Addison-Wesley Professional.

[9] Recall the definition of interaction contract: an interaction contract declares what interactions a given component can perform, expressing in high-level terms both data and control-flow constraints. Concretely, they define the condition under which a component must be activated, and which other components it then may query. Interaction contracts are defined in Section 3.2.

instance, modifications to shared code in the framework, such as the publication and subscription infrastructure, can be made in the compiler and will then take effect the next time the framework is regenerated. Where traditionally a new version of a framework would be distributed, here one would distribute a new declaration compiler to the application developers.

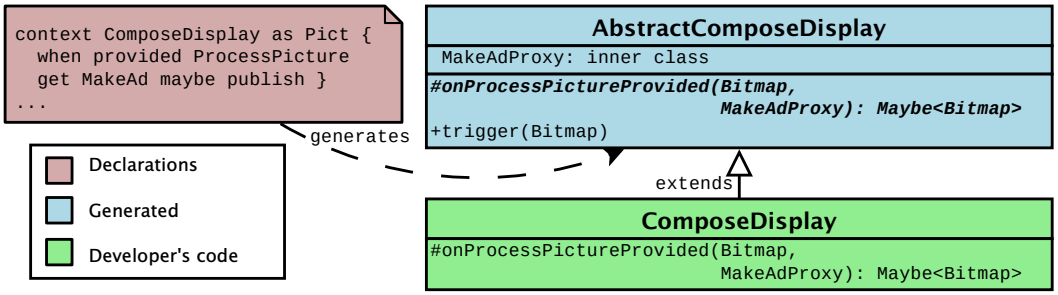


Figure 14: Schematic design of the Java prototype. Note the derived names in the generated abstract class, top right.

5.2 *Translation of the declarations*

We present the translation of the declarations into Java programming artefacts following the cases of the grammar given in Figure 4. These translations constitute the concrete instantiation of the semantics presented in Section 4.3, with the functions and holes translated into Java abstract methods which need to be filled in by extension of abstract classes. Declaration of a context or controller *C* results in an abstract class named `AbstractC`, containing one abstract method. The type of the method is derived from the interaction contract.

The return type of contexts is determined by the type annotation of the corresponding declaration—*e.g.*, `String` is the return type of the *`MakeAd`* context, as defined in Figure 5, line 7.

In the case of controllers, the return type of the abstract method is always `void`, since controllers do not compute values, but call action methods, as in the example of Figure 5, line 20. That is to say, their side-effects are important, and they should not return any values.

We will now explain the construction of the header of the abstract method case-by-case, following the Core DiaSpec grammar.

Activation conditions

The name and first parameter of the abstract method depend on the conditions under which it is supposed to activate—*e.g.*, whether a value was published, or if the component is activated by a request for data.

`when provided` ***`x`***. The abstract method will be named `on`*`x`*`Provided`.

The first argument will have the same type as x. For example,

when provided *ProcessPicture*

produces

`onProcessPictureProvided(Bitmap p, ...)`. The type of the first argument is `Bitmap`, because in the specification, *ProcessPicture* is declared to have that type (see Figure 5, line 1).

when required. The abstract method will be named `whenCRequired`, without an argument. This is because activation is not as a result of a publication of a value, but a pull request, therefore there is no value to provide as an argument yet. For example, the abstract method created in the `AbstractMakeAd` class will have a header

`String whenMakeAdRequired(...)`,

since *MakeAd* is declared to have the type `String`, and its interaction contract is **when required**.

The ellipses indicate that the rest of the method headers depend on the other arguments of the interaction contract. These are generated as follows.

Data sources and actions

These result in a tailored proxy being passed to the method, managing access to resources. They are described in the next section.

get *x*, **do *x***. Adds an inner proxy class to the abstract class, containing run-time access control. An instance of this proxy is added as an argument to the `whenRequired` or `on...Provided` method, giving the developer managed access to x. For example, the **get** *IP* declaration in Figure 5, line 9 creates the inner class `IPProxy` in *AbstractMakeAd*. Therefore, the full header for the *MakeAd* component is

`String whenMakeAdRequired(IPProxy discover)`.

In the case of *ProcessPicture*, it has no data requirement, finalising the abstract method as-is:

`onProcessPictureProvided(Bitmap p)`.

Actions for controllers are handled the same way. This proxy ensures that calls to data requirements are only fulfilled if the developer calls the proxy while the particular component is being invoked. Outside this time interval, requests are denied.

Publication requirements

These determine the return type of the method. They only apply to contexts, because controllers do not return values. Furthermore, they do not apply to contexts with a **when required** contract, as explained previously.

always publish. The return type is simply the type of the context. The types in the specification language are trivially mapped to Java

types, such as `Bitmap` for pictures.

`maybe publish`. The type of the context is wrapped in an option type. In Java we model this using `Maybe<T>`, where *`T`* is the output type. See Appendix A.2 for the definition of the option type.

An example of this is the *`ComposeDisplay`* context, which is declared as **`maybe publish`** in Figure 5. It is given the abstract method header

```
Maybe<Bitmap> onProcessPictureProvided (Bitmap p, MakeAdProxy discover),
```

since it may optionally publish a `Bitmap` value, and it is activated by `ProcessPicture` (hence the method name). The arguments result from the type of `ProcessPicture` as the first argument, and the data requirement **`get`** `MakeAd` as the second argument.

Methods `trigger()` and `notify()` are generated to map the generically-named calls in the framework to customised names such as `onProcessPictureProvided()`. See Figure 17, line 7, and Figure 18. This way we can use a generic layer of code to execute any given scenario, only generating a small amount of specialised glue code per application, which dispatches the generic calls. Note that this section presents a Java implementation of the semantics defined in Section 4.3, modelled in Java using template classes instead of lambda calculus.

5.3 *Implementing the example application*

The `ComposeDisplay` context is supposed to superimpose downloaded promotional text onto the captured image, and publish the composite image on success. The advert is downloaded from the Internet by the *`MakeAd`* context, simulating a component responsible for downloading promotional text. The developer's code is presented in Figure 15.

Here we see that the developer overrides the single abstract method, `onProcessPictureProvided`, whose type corresponds to *`ComposeDisplay`* from Figure 5, translated into a Java type as described in the previous section.

Note that we could have avoided generation of abstract classes in favour of providing classes using Java Generics,[10] for example instructing the developer to implement a class which inherits from a non-tailored class `Context<Maybe<Bitmap>>`. However, we would lose the descriptive power of generated method names, as well as the simplicity of the resource interface. The screenshot in Figure 16 illustrates how the developer is prompted with a list of possible methods to invoke on the proxy object (in this case there is only one, namely `queryMakeAdValue()`). Our technique allows the two-fold advantage of (1) allowing the developer to use familiar tools, while (2) allowing the framework to guide them.

[10] Bill Joy et al. (2000). *Java™ Language Specification*. Addisson-Wesley

```
public class ComposeDisplay extends AbstractComposeDisplay {

  @Override protected Maybe<Bitmap>
       onProcessPictureProvided(Bitmap p, MakeAdProxy discover) {

    String ad = discover.queryMakeAdValue();
    if(ad == null || ad.equals("")) {
      // problem retrieving advert, do not publish.
      return new Nothing<Bitmap>();
    }

    // ..do magic with image, overlay ad text..
    Bitmap newImg = Bitmap.createBitmap(...);

    /* manipulate newImg, code elided... */

    // publish new composite image
    return new Just<Bitmap>(newImg);
  }
}
```

Figure 15: The implementation of the `ComposeDisplay` context.

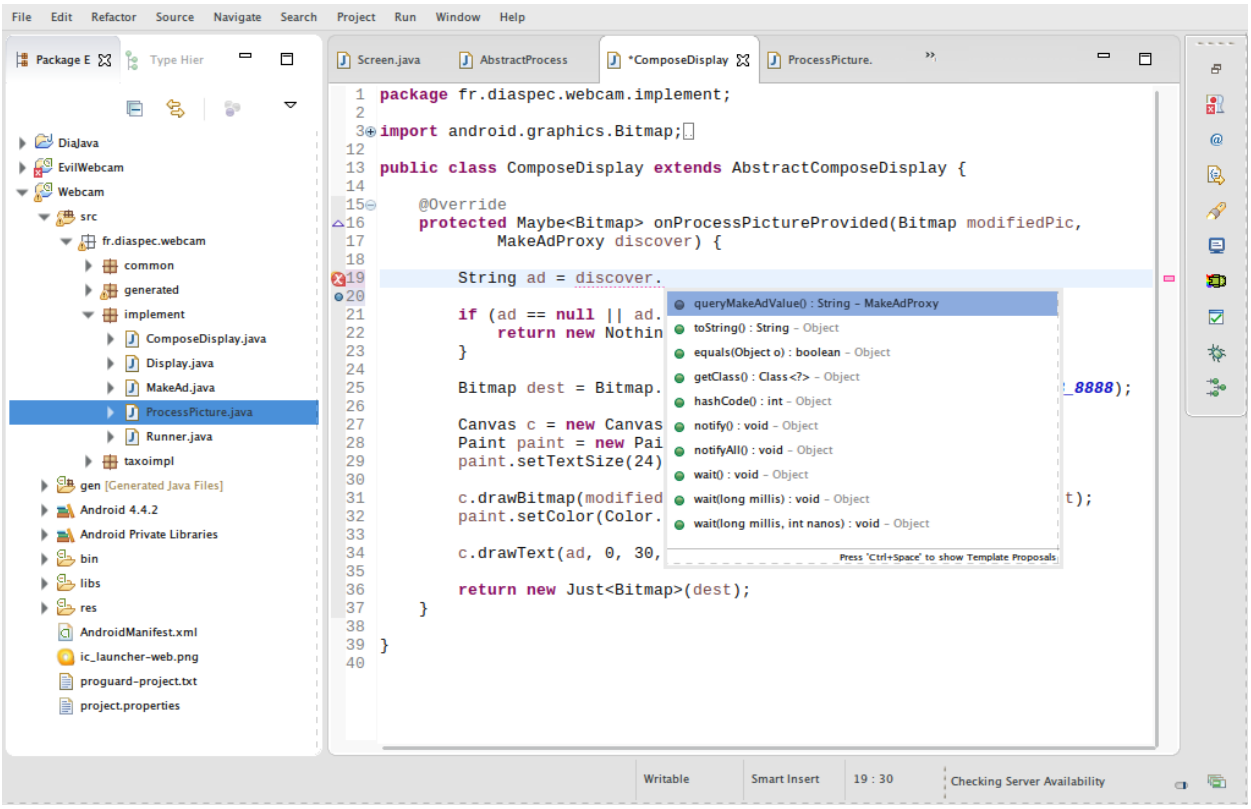


Figure 16: Screenshot of the Eclipse Java IDE, showing tailored suggestions based on declarations.

The proxy classes

Looking further at the mechanisms supporting the developer, we draw the reader's attention to the generated abstract class shown in abbreviated form in Figure 17. The `MakeAdProxy` argument comes from the declaration **`get`** `MakeAd` (Figure 5, line 14). Using the access modifier `private` and a run-time check, we ensure that *`MakeAd`* is only accessible while the framework polls *`ComposeDisplay`*. This proxy is intended to provide access restriction, plus a simpler API. A naïve approach might be to pass the result of *`MakeAd`* by value, but our approach prevents unnecessary preemptive polling, thus only calling *`MakeAd`* if the developer decides they need to. This could also pose a problem if polling has side effects. The attentive reader will notice that in line 24 of Figure 17, `requireValue()` has no argument, although according to what was previously said about data sources, it should have a proxy argument to be able to query *`IP`*. This is still true, but the inner proxy class is generated in `AbstractMakeAd`, and therefore not visible in `AbstractComposeDisplay`. Just as in Figure 17, the proxy class is generated in the class which needs access to the source. The method `requireValue()` is generated in `AbstractMakeAd`, and shown in the extract in Figure 18. Note that thanks to our approach, *`MakeAd`* has no access to the picture the user has taken.

Figure 17: Excerpt of `AbstractComposeDisplay`, including the `MakeAd` proxy.

```
1  public abstract class AbstractComposeDisplay
2    extends Publisher<Bitmap> implements Context, Subscriber<Bitmap> {
3
4      protected abstract Maybe<Bitmap>
5          onProcessPictureProvided (Bitmap newValue, MakeAdProxy localMakeAdProxy);
6
7      public final void trigger (Bitmap value) {
8        MakeAdProxy localMakeAdProxy = new MakeAdProxy();
9        localMakeAdProxy.setAccessible(true);
10       Maybe<Bitmap> v = onProcessPictureProvided(value, localMakeAdProxy);
11       localMakeAdProxy.setAccessible(false);
12       if (v instanceof Just)
13           notify(((Just<Bitmap>) v).just_value);
14     }
15
16     protected final class MakeAdProxy {
17       private MakeAdProxy ()   { }
18       final private void setAccessible (boolean isAccessible) {
19         this.isAccessible = isAccessible;
20       }
21       private boolean isAccessible = false;
22       final public String queryMakeAdValue () {
23         if (isAccessible) {
24           return runner.getMakeAdInstance().requireValue();
25         } else {
26           throw new RuntimeException("Access forbidden for MakeAd source");
27         }
28       }
29     }
30   }
```

```
public abstract class AbstractMakeAd
       extends Publisher<String> implements Context {
  protected final String requireValue ()
  {
    IPProxy localIPProxy = new IPProxy();
    localIPProxy.setAccessible(true);
    String v = whenMakeAdRequired(localIPProxy);
    localIPProxy.setAccessible(false);
    return v;
  }
  ...
  protected final class IPProxy { ... }
}
```

Figure 18: Excerpt of `AbstractMakeAd` showing the inner proxy class, which gives access to the Internet.

Binding implementations to components

Finally, to ensure all components are implemented exactly once, we also generate a class `AbstractRunner`, taking care of linking declared names to implementations. The usage of this class is shown in Figure 19. The abstract class defines methods like `getProcessPictureInstance()`, `getMakeAdInstance()`, *etc.* for a developer to override, where they should return an instance of the class implementing each component. Since `AbstractRunner` also contains the framework `main()` method, the developer is obliged to provide all the component bindings before being able to run the application. Note the use of class fields such as `processPicture` which ensure that only one instance of the implementation is executed. This is also known as a Singleton Pattern.[11] This slight inconvenience to the developer could be mitigated by introducing Java annotations; this potential improvement is discussed in the following section.

[11] Gamma et al. 1994

```
public class Runner extends AbstractRunner {

  private AbstractProcessPicture processPicture;

  @Override
  public AbstractProcessPicture getProcessPictureInstance() {

    if(processPicture == null) {
      processPicture = new ProcessPicture();
    }
    return processPicture;
  }
  ...
}
```

Figure 19: Example of deployment and binding of implementations to names from the specification. All but one of the instantiations have been elided.

5.4 *Evaluation of conformance to requirements*

Reflecting on the requirements from Section 2.3, we see that our prototype conforms.

[**Req1**: transparency] This requirement is covered by the fact that the user must validate the specification before executing the application.

[**Req2**: containment] Resource access is strictly controlled by the framework, and is only possible via the generated proxy classes which are given to the developer's code as function arguments.

The framework polls sources and publishes values, and manages the control flow. The only way to use the framework is by calling the `main()` method, which is only available after extending `AbstractRunner`. This necessitates providing well-typed implementations of all declared components.

[**Req3**: support] For the developer, implementation is simple. The API is concise and specialised, it consists of arguments passed to the implementation, nothing else. Publication is transparent, and there is no way to omit an implementation for a component.

[**Req4**: conformance] All the properties can be checked at compile-time, except for the access to data requirements. This is checked dynamically, for each access (Figure 17, line 23). This could potentially have been solved by using a Java extension with a more expressive ownership type system,[12] but this is left as future work.

[12] Nicholas Cameron and James Noble (2010). 'Encoding ownership types in Java'. In: *Objects, Models, Components, Patterns*. Springer, pp. 271–290

Limitations

One possible attack on this system could be to use some unsupervised call, such as writing to a file with a preshared name for unauthorised communication, or performing shell executions. Importing libraries also poses a threat: singleton classes might be used for unwanted communication. In fact, libraries might allow execution of arbitrary code. However, the widely deployed Android framework demonstrates that it is feasible to restrict system calls, and we could trivially analyse the use of `import` keywords in developer code. Particularly, if we allow importing the reflection library, behaviour would be very difficult to control, since developers could arbitrarily modify the control flow or call methods in classes they are not intended to have access to. Luckily it is trivial to lexically analyse the implementation classes and ensure that `import java.lang.reflect.*` and similar statements are not present. We do not restrict system calls in our prototype, but this is an avenue to explore, sandboxing each component in separate threads, and enforcing Android-like system call restrictions for each one.

Another potential threat is the use of global variables, that is, class variables declared as `public static`. This way, a developer creates a shared memory channel. As with the use of imported libraries mentioned above, this too can be easily detected and prevented by a static analysis, but that remains to be implemented in the future. As stated before, we currently make use of an unmodified Java compiler, since we considered this type of practical issue to be irrelevant to the validity of the approach.

Extensions

We might improve our prototype by defining custom Java annotations,[13] for example `@ProcessPic`. These could be used to mark which class is the implementation of which declared component. This approach could be used to generate above-mentioned abstract classes as needed when an annotation is found in an implementation class. The annotations could also be used to automatically generate the Singleton Pattern shown in Figure 19. We emphasise that this is an optional improvement to what we have presented here, since functionally it would be identical. Our solution is no weaker in terms of containing the control flow than a potential solution using annotations. We already check conformance statically, and this would not change if we used annotations. The only difference would be that the developer need not extend `AbstractRunner` as in Figure 19, as all it does is link specified components to their implementations.

[13] ORACLE (2015). *Java documentation: Annotations*.

Further discussion about possible improvements to the runtime library is provided in 'Related work', Chapter 9. Notably the discussion about capability-based systems specifically addresses the limitations discussed above. The discussion on future work in Section 10.2 additionally suggests avenues for improving our work, which includes the Java implementation.

6
Instantiation of the methodology in Racket

In this chapter, we elucidate the functional prototype. It provides the same level of support and constraint as the object-oriented prototype. We start with a general description of Racket, then proceed with the following structure: a *general description* of the design of the prototype framework in Section 6.1, the *translation* of the declarations into programming language constructs in Section 6.2, an *example implementation* of a context in Section 6.3, an additional section detailing some framework internals in Section 6.4, and finally an *evaluation* of the prototype with respect to the requirements in Section 6.5. First we present the features of Racket that are used in our prototype.

About Racket

Racket,[1] formerly known as PLT Scheme, is a dynamically-typed functional language from the Lisp language family. It has powerful syntax transformers called hygienic macros[2] that allow the creation of language extensions or even entire languages as libraries.[3] These new languages may have full or restricted use of the features of Racket. It also has an advanced module system, supporting submodules and arbitrarily many transformer stages.[4] We use these module system features to encapsulate components. The transformer stages are pre-runtime computation stages, which allow us to do static checking of specifications. Finally, a library of run-time function contracts is available.

Contracts[5] are a language extension to annotate functions with arbitrary run-time checks on input and output, with a blame system. An example of a contract, which is not to be confused with the interaction contracts of DiaSuite, is `(-> int? bool?)`, which denotes that a function must take an integer and produce a Boolean. The last argument to the `->` contract constructor is always the expected output type of the function, while the preceding terms denote the arguments.

Although Racket offers a statically typed language extension called Typed Racket,[6] we deliberately implement the functional prototype without the static typing features. Typed Racket would allow us to achieve static checks of interaction contracts, but implement-

[1] Flatt and PLT 2010

[2] Eugene Kohlbecker et al. (1986). 'Hygienic macro expansion'. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, pp. 151–161

[3] Sam Tobin-Hochstadt, Vincent St-Amour et al. (2011). 'Languages as libraries'. In: *ACM SIGPLAN Notices*. Vol. 46. ACM, pp. 132–141

[4] Matthew Flatt (2013). 'Submodules in Racket: You Want It when, Again?' In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. GPCE '13. Indianapolis, Indiana, USA: ACM, pp. 13–22.

[5] Christos Dimoulas et al. (2011). 'Correct blame for contracts: no more scapegoating'. In: *ACM SIGPLAN Notices*. Vol. 46. ACM, pp. 215–226

[6] Sam Tobin-Hochstadt and Matthew Flatt (2007). 'Advanced macrology and the implementation of Typed Scheme'. In: *In Proc. 8th Workshop on Scheme and Functional Programming*. ACM Press, pp. 1–14

ing a framework which conforms to the requirements in a statically typed language has already been demonstrated in Chapter 5. We therefore deliberately illustrate a dynamic solution. This allows us to explore the design space as far away from the previous implementation as possible, to best clarify the impact of the paradigm on the guarantees provided.

6.1 *Overview of the implementation*

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. The embedded DSL approach was chosen on the grounds of simplicity; the syntax transformers provided by Racket afford an easy way to write a compiler, which is simply a transformer defined for DSL terms. It should include constructs for defining contexts and controllers, according to the grammar in Section 3.2. As illustrated in Figure 20, 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.

Another design choice might have been to allow the developer to place context and controller implementation terms directly inside the specification. The decision was made to keep the specification and implementation separate, to allow a static check to validate the specification before allowing the implementations to be provided.

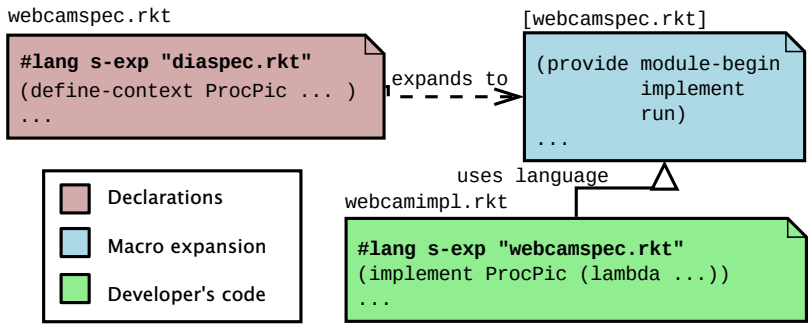


Figure 20: The architecture of the prototype. Provided declarations are transformed into a tailored language for the implementation. The `implement` macro gets cases for each declared component. The square brackets indicate the result of module expansion.

Note that the external compiler necessary for the Java prototype is entirely replaced by syntax transformers in this implementation. Transformers are not crucial to our approach, but merely make this solution more integrated. Conceptually, the two prototypes are equivalent, hence the similar visual syntax compared to Figure 14. There is a clear parallel between the way that in both prototypes, the specifications (red box) are compiled into an application-tailored framework (blue box), which is intended to be extended by

the developer (green box).

Figure 21 shows the specifications for our example application, transcribed into the Racket prototype. The syntax is a Scheme-flavoured version of the Core DiaSpec declaration language previously introduced. The presented specification mirrors the graphical representation of the application in Figure 3. It follows the grammar as presented in Section 4.2 for the Redex model, which is duplicated in Appendix A.1.

Figure 21: Complete declarations of the example application, in Racket prototype.

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

;; the sources and actions are defined elsewhere, in a central
;; taxonomy maintained by the platform owner.
(taxonomy "taxo.rkt")

(define-context MakeAd
  as String [when required (get IP)])

(define-context ProcessPicture
  as Picture [when provided Button
               (get Camera)
               always-publish])

(define-context ComposeDisplay
  as Picture [when provided ProcessPicture
               (get MakeAd)
               maybe-publish])

(define-controller Display
  [when provided ComposeDisplay
    do Screen])
```

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

6.2 *Translation of the declarations*

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. This is similar in spirit to the Common Lisp Object System (CLOS)[7] approach, which provides `defgeneric` and `defmethod`. There, `defgeneric` allows one to declare a generic function, and `defmethod` allows one to provide a specialised implementation. Signature and implementations are decoupled.

[7] Daniel G. Bobrow et al. (1988). 'Common Lisp Object System Specification'. In: *SIGPLAN Not.* 23.SI, pp. 1–142.

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 expression f as the implementation of C. For example, a developer may write `(define-context C1 as ...)`, and implement C_1 using `(implement C1 (lambda (..) ...))`. An example of this is shown in the diagram in Figure 20. The following section describes the precise relationship between definitions and implementations.

However, not just any f may be provided, as the arguments to **`implement`** are subject to a function contract.[8] These function contracts are the direct result of interpreting the semantics presented in Section 4.3. Contracts in Racket are flexible annotations on definitions and exports, which perform arbitrary tests 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 DiaSuite interaction contracts of Figure 21 as follows.

[8] Dimoulas et al. 2011

Activation conditions

These define the first argument to the function f, and thus the first argument to **`->`**, the function contract builder.

`when provided x`. First argument gets type of x. For *`ComposeDisplay`*, the contract starts with `(-> bitmap%? ...)`,[9] since it is activated by *`ProcessPicture`* publishing a bitmap image. The percent symbol at the end of `bitmap%` signifies that it is a class; in Racket all class names end with `%` by convention.

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

`when required`. No argument is added for the activation condition—the context was activated by pull.

Data sources and actions

These determine the (optional) next argument to the developer's function. This argument is a closure providing proxied access (that is, surrounded by a run-time guard or dynamic check) to the resource. This preserves convenience for a developer querying a resource, and allows the framework to enforce permissions. Actions for controllers are provided using the same mechanism.

`get x`. The contract of the proxy closure is `(-> t?)` where t is the output type of x. As an example, consider the *`ProcessPicture`* context. It is activated by a button which we model as a Boolean value, and has a `(get Camera)` declaration. The full contract for *`ProcessPicture`* so far is therefore `(-> boolean? (-> bitmap%?) ...)`. The ellipsis refers to the rest of the contract, defined next.

`do x`. This clause can only appear inside controller declarations. The contract of the proxy closure is `(-> t? void?)` where t is the input type of an action x—this reflects an action with argument of type t, which does not return a value. The full contract for a

controller is therefore `(-> ... (-> t? void?) void?)`. The final `void?` reflects that controllers do not return values. Continuing the previous example, the full contract for *`Display`* is therefore `(-> bitmap%? (-> bitmap%? void?) void?)`.

Publication requirements

The publication requirements determine the last argument(s) to the function contract of a context. This corresponds to the output type of the implementation term. Publishing is handled using continuations, which give us flexibility in the number of 'return' statements provided. The choice for continuations is arbitrary; we could also model this behaviour with an option type, for example. However, providing a continuation allows us to prevent control from returning to a context once it has published. Continuations allow very explicit management of the control flow.

Note that the `none/c` function is a utility contract provided by the Racket base library. As the name suggests, it accepts no values: it causes a run-time exception to be thrown if control reaches the end of a function with such a contract. This happens if the developer does not use one of the provided continuations, which are the only acceptable methods to return control to the framework.

`always-publish`. If a context must always publish, one continuation function is provided, corresponding to publication. The final contract becomes
`(-> ... (-> t? void?) none/c)`,
with t the expected return type. For example, the full contract for the *`ProcessPicture`* context is
`(-> boolean? (-> bitmap%?) (-> bitmap%? void?) none/c)`.

If the developer does not use the publication continuation (the third argument) an exception will be raised by the `none/c` contract.

`maybe-publish`. If publication is optional, two continuations are passed to f, for both the publish and no-publish case. The first has the contract `(-> t? void?)` with t the expected output type. The second continuation simply returns control to the framework without publishing a new value. If the developer chooses not to publish, they use the second, no-publish continuation. The contract is therefore
`(-> ... (-> t? void?) (-> void?) none/c)`.

The *`ComposeDisplay`* context provides a concrete example; it has the contract
`(-> bitmap%? (-> string?) (-> bitmap%? void?) (-> void?) none/c)`.

We see an example of the use of these continuations in the following section—the example in Figure 22 illustrates the use of two continuation functions.

6.3 *Implementing the example application*

```
1  ;;; Implementation file, webcamimpl.rkt
2  #lang s-exp "webcamspec.rkt"
3
4  (implement ComposeDisplay
5    (λ (pic getAdTxt publish nopublish)
6      (let* ([canvas  (make-bitmap 450 450)]
7             [adTxt   (getAdTxt)]
8             [dc      (new bitmap-dc% [bitmap canvas])])
9
10       (cond [(string=? "" adTxt) (nopublish)])
11
12       ;; ... do magic, overlay adTxt on pic
13       (send dc draw-bitmap pic ..)
14       (send dc draw-text adTxt ..)
15
16       (publish canvas))))
17
18 ;; the remaining implement-terms
19 ...
```

Figure 22: The implementation of the `ComposeDisplay` context. The developer creates a new canvas and paints the received image `pic` onto it, followed by the text `adTxt`.

In Figure 22, 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, *i.e.,* Figure 21, line 16. The implementation a developer provides should be in the form of a lambda expression obeying the contract resulting from the specification. For example, the `ComposeDisplay` context has the contract
`(-> bitmap%? (-> string?) (-> bitmap%? void?) (-> void?) none/c).`

The contract has this structure because the *`ComposeDisplay`* context is activated by `ProcessPicture` publishing an image, it has **`get`**-access to the `MakeAd` component which returns a string, and the context may or may not publish an image on account of the **`maybe-publish`** specification. The last two continuation arguments correspond to publishing a value `(-> bitmap%? void?)` and not publishing `(-> void?)`. The lambda expression provided by the developer in Figure 22 conforms to this contract. Note that it is possible to refer to either or both of the continuations in the implementation code, but it is impossible to execute both. This is because the framework regains control when a continuation is called and control is never passed back to the context. We see that if the advertisement component returns an empty string (line 10) the developer decides not to publish, but otherwise the string is overlaid on the picture and the developer publishes the composite image (line 16).

Encapsulation

To prevent implementations of different components communicating outside of the condoned pathways, the `implement` macro wraps each provided implementation f in its own submodule. As illustrated in Figure 23, lexical scoping prohibits these functions

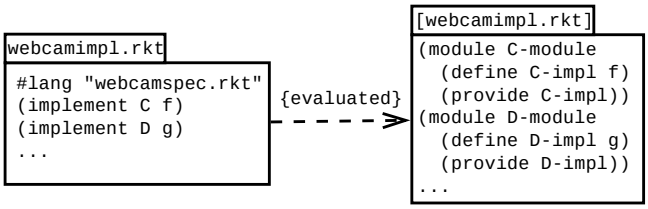


Figure 23: Separation of components using modules. The developer's code (left), and its expanded form (right). The function f in C cannot access D or g, because of lexical scoping.

from accessing their surrounding terms. Only the implementation f is directly exported for use in the top-level implementation module written by the developer. The resulting Racket code after this wrapping is shown in Figure 24. The code in grey is precisely the term provided in Figure 22, but it is now isolated from the implementations of the other components, preventing leaks caused by the developer being able to access them. Note how the previously constructed contract has been attached to the developer's implementation term, and the fact that only the `C-impl` terms are exported back to the `webcamimpl.rkt` module.

```
(module ComposeDisplay-module racket/gui
  (define/contract ComposeDisplay-impl
    (-> bitmap%? (-> string?) (-> bitmap%? void?)
        (-> void?) none/c)
    (λ (pic getAdTxt publish nopublish)
      (let* ([canvas  (make-bitmap ..)]
             [adTxt   (getAdTxt)]
             [dc      (new bitmap-dc% [bitmap canvas])])

        (cond [(string=? "" adTxt) (nopublish)])

        ;; .. do magic, overlay adTxt on pic
        (send dc draw-bitmap pic ..)
        (send dc draw-text adTxt ..)

        (publish canvas))))
  (provide ComposeDisplay-impl))

;; remaining implementations
...)
```

Figure 24: The developer's code snippet is encapsulated in a submodule, as a result of evaluating Figure 22. The shaded code is simply the term provided by the developer, which has been spliced into a new submodule.

Note that alongside this snippet, the rest of the implementations of the declared components are provided in the same module. This module must be implemented using the new `webcamspec.rkt` language—the one resulting from the expansion of the specification terms the developer has written. The implementation module is checked to contain exactly one `(implement C ...)` term for each declared component C; shadowing is not allowed. We focus on a single context implementation to illustrate what transformations are done on the developer's code. These transformations notably arrange for the publication system to be able to deliver messages correctly, and to isolate the implemented components from one another.

When the developer has provided implementations for each of the declared components, they can evaluate the implementation

module, and use the `(run)` convenience function that is exported by the module resulting from the specifications. In the next section, we describe how the macros which add contracts and modules are implemented.

6.4 *The framework and run-time*

Now that we have seen the user interface for our framework—*i.e.*, that which the application developer deals with—we elucidate how the framework internals are implemented. The framework 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) the run-time support libraries that tie the implementations together to provide a coherent system. These mechanisms are explained globally here, though certain implementation details are elided. Notably, making all the identifiers we introduce available in the correct transformer phases and module scopes was intricate. We invite the reader to experiment with the prototype code—the functionality for (1) and (2) is in the `diaspec.rkt` module, while (3) the run-time library can be found in the `fwexec.rkt` module.

Syntactic transformation of the declarations

Previously we saw that the first step for the application developer is to declare the components of an 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. This tag causes the entire syntax tree of the specification file to be passed to the function **`#%module-begin`**, exported from `diaspec.rkt`. 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 implementations.

To illustrate, Figure 25 shows the expansion of the `ComposeDisplay` declaration, from line 16 of Figure 21. The expansion corresponds to the blue box, top right, of Figure 20. Simplifications have been made, and module imports, *etc.* have been omitted for brevity. Some terms which are not influenced by this declaration term have been elided, for example a helper macro which transforms terms of the form `(implement x ...)` into `(implement-x ...)`. This helper is included in Appendix A.3. Its purpose is to translate between the syntax the developer uses to provide implementation terms, and the generated macro in line 34 of Figure 25. Also not shown here is the 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 code listing (see line 54), since it is not tailored per application. It can be found in Appendix A.3. Note that it is this definition which allows the

Figure 25: The simplified expansion of the specifications, concentrating on `ComposeDisplay` from Figure 21.

```
(module webcamspec "diaspec.rkt"

 ;; internal representation of interaction contract
 (define ComposeDisplay
   (context 'ComposeDisplay
     (interactioncontract ProcessPicture MakeAd
       'maybePublish) 'pic))
 (provide ComposeDisplay)

 ;; add a contract to the contracts module
 (module+ contracts
   (define ComposeDisplay-contract
       (-> bitmap%? (-> string?)
           (-> bitmap%? void?) (-> void?) none/c))
   (provide ComposeDisplay-contract))

 ;; a specialised struct to hold the implementation
 (define-struct/contract ComposeDisplay-struct
   ([spec   (or/c context? controller?)]
    [implem (-> ...)])) ; same contract as line 13, above
 ;; the specification module exports the implement-.. functions:
 (provide ComposeDisplay-struct
          implement-ComposeDisplay)

 ;; syntax transformer which wraps the term provided by the developer in
 ;; its own submodule. for example:
 ;;
 ;; (implement f (lambda ..))
 ;;      ~~>  (roughly)
 ;; (module f-submodule
 ;;   (provide f-impl)
 ;;   (define/contract f-impl f-contract f))
 ;;
 (define-syntax (implement-ComposeDisplay stx)
  (syntax-case stx (implement-ComposeDisplay)
   [(_ f)
     #'(begin
         (module ComposeDisplay-submodule racket/gui
           (require (submod "webcamspec.rkt" contracts))
           (provide ComposeDisplay-impl)
           (define/contract ComposeDisplay-impl
                            ComposeDisplay-contract f))

         (require (submod "." ComposeDisplay-submodule))

         (set-impl 'ComposeDisplay ; add to hashmap
           (ComposeDisplay-struct ComposeDisplay
              ComposeDisplay-impl)))]))

 ... ;; other contexts omitted

 (provide run (rename-out (module-begin-inner #%module-begin)))

 (define-syntax (module-begin-inner stx2)
    ... )) ;; module-begin-inner elided, see Appendix A.3
```

specification module to be used as the implementation module language, using the `#lang s-exp "webcamspec.rkt"` directive.

Line 1 of Figure 25 marks the start of the expanded specification module, called `webcamspec`. It still references `diaspec.rkt`, the language of the specification, see Figure 21. The rest of the displayed code results from the declaration of the `ComposeDisplay` context. In line 4, we see that a binding is introduced, using the name the developer chose for the component. The value of `ComposeDisplay` is a representation of the declaration, and is used to derive the contract. In line 11, a submodule is guarded by 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 18 defines a tailored struct: it will hold the implementation of `ComposeDisplay`, in the `implem` field tagged with the corresponding contract. The complexity increases in line 34, 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 39.

[10] Flatt 2013

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.[11] Crucially, it also substitutes `f`, the pattern variable bound by **`syntax-case`** in line 36, with the pattern variable's match result, in this case the developer's implementation term.

[11] See the Racket documentation for more information on the **`syntax`** keyword. Matthew Flatt and PLT (2015a). 'Pattern-based Syntax Matching'. In: PLT-TR-2010-1. `http://racket-lang.org/tr1/`, Version 6.2.1. Chap. 12.1.

Tying it all together. Next, in line 44, 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 46). 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.

Because of where it is defined, the `run` function is only available to the developer if they can evaluate the implementation module without syntax errors, which implies that only valid combinations of specifications and implementations allow the developer to execute the framework. Since the implementation of the framework run-time library is straightforward, we do not discuss it here. To run this code, DrRacket[12,13] can be used, or any other Racket environment, such as `racket-mode` for Emacs. Simply load the `webcamimpl.rkt` file, and when it is loaded, evaluate `(run)` in the REPL.

[12] Robert Bruce Findler et al. (2002). 'DrScheme: a programming environment for Scheme'. In: *J. Funct. Program.* 12.2, pp. 159–182.

[13] Tested using DrRacket v6.1.1.

The development environment. In Figure 26 we present a typical DrRacket session, showing syntax highlighting for one of our specification files. Notice how DrRacket can point out the location of binding sites for components, thanks to the fact that the syn-

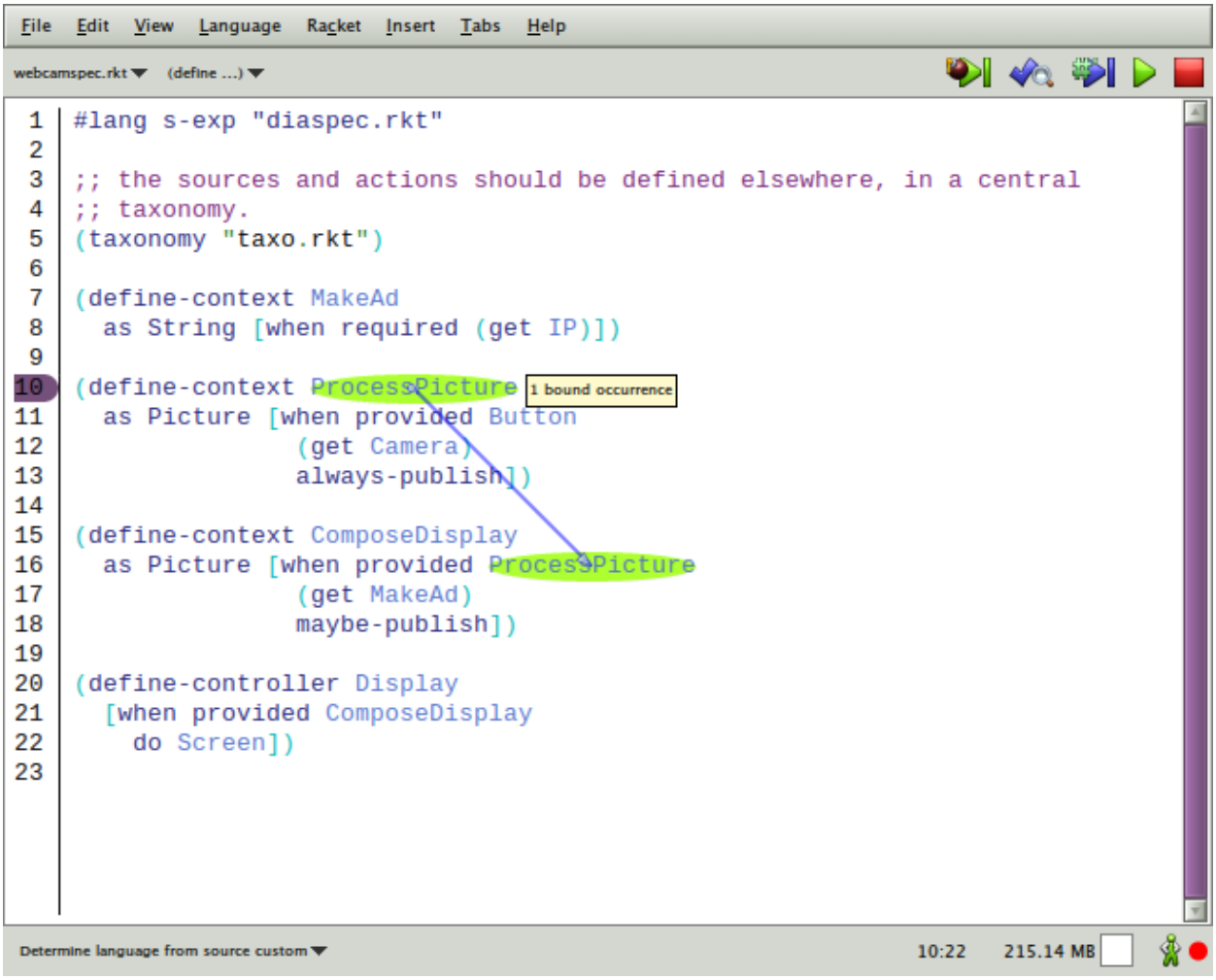


Figure 26: Unmodified screenshot of the DrRacket GUI, indicating available binding information. Note the blue arrow in the code window, pointing from the binding site of the `ProcessPicture` term (line 10) to where it is used (line 16).

tax transformers preserve lexical information. This illustrates the strength of the language extension approach: we can add or restrict existing languages using Racket, without having to sacrifice the power of existing tools which have already been developed for Racket.

6.5 *Evaluation of conformance to requirements*

Reflecting on which language mechanisms have been used to implement the various requirements, and how well they are met, we summarise as follows.

[**Req1**: transparency] This requirement is covered by the fact that the user must validate the specification before executing the application.

[**Req2**: containment] Resource usage is controlled by the framework. The developer's code is isolated using submodules and only given access to resources via checked proxy closures.

By providing continuations as proxies, which check publications,

we can be sure that the developer cannot influence the control flow. Combined with submodule scoping this ensures encapsulation of components.

[**Req3**: support] Thanks to the novel use of a tailored language extension, the implementation does not burden the application developer. The developer is provided with helpful syntax errors if for example an implementation is omitted or the specification is incoherent, and the API merely consists of the allowed resources being passed to the implementation as function arguments.

[**Req4**: conformance] The structure of the implementation is verified statically, but the types of values a developer provides are dynamically checked using contracts.

All properties resulting from the specifications are checked and enforced. We ensure the same level of restriction as the Java prototype, and have managed to give the developer a clear and concise way of implementing the components. There is no complex API to communicate with other components, and the verification is mostly static. The types of values are checked using Racket contracts, at run-time.

Flexibility of implementation language. If we switched to Typed Racket,[14] the checks would be static, like in Java. Switching to Typed Racket would be trivial—it amounts to changing the language to `typed/racket` instead of `racket/gui` (in Figure 24, on line 1), and translating the contract syntax on line 3 into the very similar Typed Racket syntax. This is a strong point of Racket: allowing us to easily use the right language for each module, gluing them together using the common Racket run-time system. We also note that it is perfectly feasible to choose different languages for different component modules. This could in principle be used to make resource access controls dynamic for certain modules, and static for others. As a concrete example where this might make sense, perhaps it would be enough to ensure at compile time that the use of the camera is authorised, but for the contact list it might make sense to check access dynamically. We could imagine that every time an application tries to load specific contact details, the framework asks the user interactively for confirmation, listing the specific information to be given to the application. An application trying to iterate over all contacts to exfiltrate them would then be detected very easily by the user.

[14] Tobin-Hochstadt, St-Amour et al. 2011

Static or dynamic checks, as needed. The strength of this prototype is that it permits fine-grained control over where exactly in the static *vs.* dynamic design space to place permission checks for a given resource. The prototype can even be used as a tool for experimenting with the feasibility and efficacy of static *vs.* dynamic checks for various resources. We do not necessarily advocate always

using static or dynamic checks, but rather whatever best fits the particular type of resource.

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 application, then glue modules together via the common run-time library provided by Racket. This allows great flexibility and control, since with Racket's **`#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.

Dealing with libraries and reflection. Concerning the separation into submodules to avoid unauthorised communication, there are ways a malicious developer might get around this restriction. For example, importing a module *M* with shared mutable state through the use of `set!` statements. Contexts *A* and *B* could both import module *M*, and use it as a communication channel. In fact, importing modules would allow arbitrary code execution. This problem is the same as described in Java. However, since the `require` keyword in Racket is only allowed at the top-level of a module, the isolation as presented should be sufficient. One way of circumventing this could be to build an expression and import the module at run-time, for example with `(`**`eval`** `'(`**`require`** `...))`. In fact, many nefarious things could be done this way. For this reason we completely disallow the use of **`eval`** and **`require`** in implementations. This is not watertight, but gives a suggestion on how to mitigate such leaks. Unfortunately by nature of the fact that Racket is a dynamic language with powerful reflection, there are potentially ways to hide the binding of **`eval`**, but we consider this threat outside the scope of this work. This highlights a need in Racket: allowing components or functions to be pure would solve this vulnerability. Perhaps Typed Racket[15] will offer a solution in the future—purity analysis is on the project to-do list.[16]

We might later consider adding a safe, vetted, way for developers to specify which libraries they would like to use, since currently only the Racket base library is provided. It would be easy to provide our own `require`-like keyword, although this might allow aforementioned unauthorised communication (breaking **Req2**). Only benign modules should be allowed to be imported, but the discussion on how to determine which modules qualify is out of the scope of this work. For now we assume that the platform owner decides which modules to allow, if any.

Alternative strategies. Note that it is not essential to use the language extension feature, or even Racket, to implement a similar framework, one could instead generate macros for a developer to use without imposing a DSL. This would be equivalent to a 'framework creation kit' as opposed to a solution for a class of applica-

[15] Tobin-Hochstadt and Flatt 2007

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

tions. Defining a language extension, however, allows full control over the implementation: we can enforce that only **`implement`** terms appear at the module level, or that the declarations only consist of **`define-context`** and **`define-controller`** terms. It also permits fine-grained control over which type of checks to apply where, in the spirit of gradual typing.[17]

We could also have refrained from using macros, simply manipulating the specification as plain data structures. Apart from the inconvenience of not having specialised syntax, this would have the disadvantage that checks such as coherence of the implementation, conformance of the implementation to the specification, *etc.* would all be dynamic. The programming interface would also necessarily not be as focused as in the prototype.

[17] Jeremy G. Siek and Walid Taha (2006). 'Gradual Typing for Functional Languages'. In: *Scheme and Functional Programming Workshop*, pp. 81–92

Limitations

There exist a number of ways in which we can improve this prototype. Firstly, we note that the chosen platform and model are merely examples, it should be feasible 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 might allow them to modify the contracts, rendering the applications unsafe. This requires more experimentation.

Conclusion

In conclusion, via this prototype we have demonstrated that it is feasible to implement the same level of developer guidance and restriction as was available before, in a dynamic functional language.

The requirements are met in an equivalent fashion when compared to the Java prototype. The limitations of our prototype that we have listed above are furthermore not fundamental in nature, but could be addressed by using various well-known approaches. These are discussed in the 'Related work' chapter, especially Section 9.4, which deals with language-level behavioural restrictions.

7
Lessons learned

After having implemented the prototypes in the previous two chapters, we can make a meaningful comparison with widely used declaration-driven frameworks and the state of the art of security for mobile computing. Furthermore, we provide principles which summarise the conclusions of this thesis.

7.1 Comparison to mainstream frameworks

In comparison with the static analysis approaches for Android, discussed in Section 2.1, our approach differs in that it requires modification of the run-time platform. We argue that this is justified by gains in terms of both privacy for the user and guidance for the developer. Attacking the problem using general-purpose static analysis techniques on unmodified applications seems more difficult a problem than strictly necessary. Rather, given our decomposition of an application into components, a more focused analysis could be performed. Furthermore, our approach would allow extraction of flow rules from the specifications, to be passed to the analysis software. These should then be checked to be respected upfront, as opposed to attempting to extract all potential data flows inside an application binary. We conjecture that our decomposition into independent components potentially allows making assumptions concerning the execution environment of each snippet of code, which would allow a less exhaustive analysis.

We refer again to the TouchDevelop approach mentioned in Section 2.1. It is different to our approach, since a developer has to learn a new language, whereas we are able to achieve meaningful and fine-grained restrictions while allowing a programmer to use their familiar general-purpose programming language, allowing freedom to choose IDE, libraries, tools, *etc.* It is simple to extract arrows such as those provided by the TouchDevelop approach from our declaration language (*e.g.*, camera $\rightarrow$ WWW). Thanks to this, our approach can provide a high degree of insight to the end-user regarding which resources might be used, and how, which is a significant advance compared to the current generation of widely used frameworks.

In Section 2.1, we also discussed the static analysis approaches

taken by Elish et al. 2013; C. Mann and Starostin 2012; Fritz et al. 2013; Gibler et al. 2011; Mirzaei et al. 2012. These approaches fail to provide an API that is as targeted to each application architecture as in our prototypes. This should prevent a lot of wasted effort for application developers, discovering which library method is needed to access a particular resource. Additionally, when developers are faced with overwhelming lists of possible permissions, research shows that they frequently select many more permissions than necessary to simply make the application work.[1] Finally, our methodology indicates checks can also be implemented statically, avoiding the potential pitfall in mainstream frameworks, such as Android or Apple's iOS, where access to a disallowed resource can cause run-time exceptions. On both iOS and Android, the permissions are enforced dynamically, even if the moment at which permission is granted differs: before run-time for Android, versus on iOS, the moment the permission is requested for the first time. Both platforms suffer from the fact that if an application is denied an expected permission at run-time, it is likely to crash. Our methodology, on the other hand, outlines how resource permissions can be statically enforced.

[1] Wei et al. 2012

7.2 *Principles*

In this section we discuss the lessons relevant to declaration-driven framework design that we have learned from the implementation of the prototypes in Java and Racket. We try to generalise the lessons from the implementations to framework design, and how requirements translate to programming features in general, to guide future implementations of such frameworks.

We have developed a system where declarations regarding structure and behaviour of an application are used to provide a programming environment which actively ensures requirements are met. At the same time, it reduces development effort for the application developer by removing much of the needed boilerplate code. We do not claim that this is the best engineering approach to implement a tailored framework; rather, we argue that declaration-driven frameworks can deliver significant advantages in a wide spectrum of programming languages.

A strong case for rich declarations

Both in object-oriented and functional settings, we see that declaration-driven frameworks are able to turn declarations from an advisory document of promises into verifiable properties. This can provide the end user with valuable information on the behaviour of an application. Also, we claim that from the developer's point of view, implementing an application using a tailored framework is less laborious than using a general-purpose framework. In our example, only the components need to be implemented. All com-

munication, deployment, *etc.* is taken care of by the framework. This is possible because the framework has detailed information about the intended structure and behaviour of the application.

Therefore, we believe that this new generation of frameworks can provide fundamental advantages. Even if general-purpose frameworks which are not tailored to each application already provide a notion of restrictions, the available declarations—*e.g.,* the `Manifest` file in Android—are currently not used to ease development. This is a missed opportunity.

We also argue that the implementations in both Java and Racket are natural. Extending classes is the traditional way of using an object-oriented framework, as is the use of domain-specific languages to solve problems in a Lisp-like language. We automate these mechanisms without imposing a steep learning curve on the developer, or requiring them to learn new implementation techniques. It is feasible to go beyond the status quo and enforce properties, give the user insight into the usage of their resources, and assist the developer.

The trade-off between static and dynamic resource restrictions

We have seen that controlling access to resources, or even more generally speaking, enforcing a certain control flow is essential to the open platform domain. This need can be fulfilled by declaration-driven frameworks. For each resource the framework developer may choose to handle the restrictions either statically or dynamically, depending on the sensitivity of the resource. It is not necessary to choose one approach globally. In fact, it makes most sense to decide per-component which approach is most suitable. This is analogous to the gradual typing approach as advocated by Siek, et al.,[2] where the developer may decide per-function whether to employ static type checking.

[2] Siek and Taha 2006

As with type systems, if a static approach is chosen, an advantage is early warning if a developer performs an invalid operation, but with the disadvantage of less accurate specifications. For example, approximations inherent to the static approach may result in requesting some permissions which could remain unused, leaving unnecessary potential for privacy leaks. Consider the case study on the other hand, where a developer cannot access unauthorised resources, without triggering compile-time errors.

If a dynamic approach is chosen, an advantage is a high degree of accuracy regarding which resources are used, and when. A user can be interactively prompted to allow or deny specific requests. However, the trade-off is receiving late warnings about incorrect code thus forgoing a degree of developer guidance.

For example, as in Apple's iOS, dynamically-handled resource access controls are the most accurate: the user can choose to allow or disallow requests on a per-resource basis, if and when access is requested. Unfortunately, the implementation still misses an

opportunity: an improvement to the iOS approach would be to ask permission for each resource request, instead of once for permanent access to the resource. This still would not give the user a clear view on what happens with sensitive data, though. It is possible that a legitimate request for sensitive data is also used to camouflage exfiltration, as discussed in Chapter 1.

Compared to iOS, permissions in Android are also dynamically checked, but even less favourably. Even if a given permission is unused for a particular session it still has to be allowed by the user at install time. This is a vulnerability: advertisement libraries routinely abuse their embedding into over-privileged applications,[3] allowing them to exfiltrate all private data the application has access to. In our approach, the permissions are also defined once for all sessions, but are specified per-component. This is an advantage: the developer can be helped by specialising the API per application, and giving warnings about misuse of resources at compile time. Our approach also allows the addition of dynamic checks, thus achieving parity with iOS, in a sense offering the best of both worlds.

[3] Wei et al. 2012; Bartel et al. 2012; and Stevens et al. 2012

The fact that a recent version of Android contains a hidden screen to (dis)allow access per-application and per-resource[4] highlights that the old permission model was not meeting users' needs. This suggests that the original design choices were ad hoc, with implicit principles. We hope that our work can clarify the possible design space for such systems. Also, it aims to provide a conceptual framework for the design and implementation of declaration-driven frameworks, helping to make informed choices between static *vs.* dynamic enforcing of particular rules, including advantages and trade-offs.

[4] *Hidden Android feature allows users to fine tune app permissions* (2013). Online, `http://www.zdnet.com/hidden-android-feature-allows-users-to-fine-tune-app-permissions-7000018944/`. Accessed: May 2015

Viability of enforcing requirements is independent of programming paradigm

We have some computations which are specific to our declaration-driven approach, such as checking whether queries to resources are legitimate. Depending on whether we choose to process the declaration semantics statically or dynamically, we get differing support and restriction.

We observe that the choice between statically and dynamically handling the declarations is orthogonal to whether the host language is statically or dynamically typed. This is shown by the identical guarantees in both the Java and Racket prototypes. In fact, in general, we find that a type system is not even a prerequisite to being able to realise all the requirements introduced in the case study.

More generally, all we need is a programming environment with at least one stage before run-time, enabling processing of the static semantics of the declarations. Consider the Racket example, where we make no use of traditionally static features or a type system, but implement the declaration compiler using syntax transformers. Syntax transformer phases can be seen as extra compilation phases.

Indeed, a syntax transformer system generalises a static type system: a type system can be considered a limited-expressiveness programming stage, as suggested by the implementation of Typed Racket.[5] In Java, this is precisely how the properties are verified. However, since Racket is a very extensible and expressive language, it might give an optimistic impression of what can be achieved in other, less expressive, host languages. This does not invalidate our results, but means that what was easy in Racket might require more engineering in other languages.

[5] Tobin-Hochstadt and Flatt 2007

Therefore, if we have stages (or can implement them, whether through a declaration compiler or macros), the place to handle the enforcing of requirements and obligations arising from the declarations, is in the stage(s) before run-time. In our case study, we used code generation plus the type system for the Java framework, and transformer phases for the Racket implementation, to achieve this kind of checking. Therefore, widely varying techniques can be used to implement stages. A strong and/or static type system is thus not required, even if static enforcing is desired.

Part IV

Conclusion

8
Generalisation of our approach

The main parts of this dissertation have shown that the methodology we present is effective in the domain of mobile computing, where it effectively addresses concerns of user data privacy. This chapter argues that the methodology is more generally applicable, and that it would provide advantages in diverse application domains, offering a range of benefits, going beyond user privacy. This chapter refers to ongoing work in the research team, which is why the discussion in this chapter will be superficial, merely arguing that the application of the methodology presented in this dissertation is but one among many possibilities.

Section 8.1 presents a direction of research involving monitoring of older adults, in the domain assisted living applications, using a methodology comparable to the one presented in this dissertation. This research is currently being pursued by others in the research team as part of the HomeAssist project.[1] Section 8.2 summarises other application domains where this methodology would be beneficial.

[1] For more information about the HomeAssist project, see http://phoenix.inria.fr/research-projects/homeassist.

8.1 Application to assisted living

The methodology presented in this dissertation guides *platform owners* (as defined in Section 2.3) to implement declaration compilers which output programming frameworks. Since this framework-producing infrastructure is in place, it is easy to add layers, for example for application monitoring, to the platform implementation.

In the context of assisted living, the platform consists of a collection of sensors installed in the homes of older adults. These sensors are the resources of our platform, which installed applications can use. These applications are typically for monitoring older adults, for example to ensure that the inhabitant has had their breakfast, and if not, raise an alert. These types of applications entail some different challenges to those in the domain of mobile computing. Typically, the user does not interact directly with the applications. Instead, they are expected to run as services in the background, with a high level of dependability.

This poses a potentially high cost of maintenance and monitor-

ing of the platform and applications by the platform owner or the caregiver who administers the platform.

Our approach makes it easy to add logging or real-time monitoring capabilities to a platform, for example by inserting instrumentation instructions in the generated code discussed in Section 5.3. This is a significant advantage for the design of an assisted living platform, where cross-cutting concerns such as logging need to be weaved into the system. The expectation is that in the future, the HomeAssist project[2] will provide a validation of our methodology in this domain.

[2] Lucile Dupuy, Hélène Sauzéon and Charles Consel (2015). 'Perceived Needs for Assistive Technologies in older adults and their caregivers'. In: *womENcourage 2015*. Uppsala, Sweden.

The LogSim project

Furthermore, given the fact that such logging layers can easily be added, this has permitted the collection of a large amount of real sensor data from HomeAssist installations. After the work done to simulate applications using our methodology with synthetic data[3] in the domain of home automation, the next step is to test assisted living applications using realistic sensor data collected from applications where instrumentation has been injected. Using this type of data, the LogSim project[4] aims to implement an assistive application simulator to check application usefulness and usability through *in vivo* logs. It aims to provide the developer with feedback at each stage of development, by automatic creation of mock input devices, based on known-good application execution logs which have been verified by an aging expert. Additionally, rules can be written to validate the dynamic behaviour of an application, opening up possibilities of real-time monitoring of applications once deployed.

[3] Julien Bruneau and Charles Consel (2013). 'DiaSim: a simulator for pervasive computing applications'. In: *Software: Practice and Experience* 43.8, pp. 885–909.

[4] The LogSim project has not yet produced published results, but more information can be found at `http://phoenix.inria.fr/research-projects`.

The aim for this project is therefore to provide a development environment which supports the full application development life cycle, from design using a specification language as presented in this work, to developer feedback through testing using log data recovered from *in vivo* assisted living installations, to deployment.

8.2 Other application domains

Assisted living and platform monitoring are just some examples of the extensions which our methodology might offer in the future. Other application domains include the generation of C programming frameworks targeted to embedded devices.[5] This would be useful in combination with the assisted living platform, which makes heavy use of embedded devices and sensors. These generated programming environments could be used to decrease development effort for such devices and increase their dependability: as a result of the low-level nature of embedded software development, it is prone to programmer error.

[5] Gilles Muller et al. (2000). 'Towards robust OSes for appliances: A new approach based on Domain-Specific Languages'. In: *ACM SIGOPS European Workshop*. France, pp. 19–24.

9
Related work

In this chapter, we provide an overview of existing approaches for restricting access to users' private data by applications. Note that Chapter 2 also contains details about various approaches, although here we focus more on making comparisons with our methodology.

We first present other approaches based on rich declaration languages in Section 9.1. Next, we present other work specifically aimed at securing mobile computing applications for the mainstream platforms in Section 9.2. Then, in Sections 9.3 and 9.4, we present other approaches of a more fundamental nature, aiming to restrict application capabilities or permissions.

9.1 Frameworks enriched with declaration languages

An example of a declaration-driven framework supported by a rich declaration language is Yesod,[1] a web framework written in Haskell. Similar to our approach, Yesod makes use of declarations to tailor the framework per-application, and then to guide the developer, statically verifying that the implementation is free of broken URLs, missing components, *etc.* The Yesod documentation does not discuss the design space and the potential benefits of the use of declarations. Therefore, it is unclear if the declarations are being put to optimal use. Also, that approach is focused on web applications, whereas our methodology is more generally applicable (see Chapter 8).

[1] Snoyman 2012

Our work is inspired by DiaSuite,[2] and is a continuation thereof, therefore the DiaSuite approach deserves special mention. However, while the articles related to DiaSuite do explain the theory of interaction contracts and the idea of a generated framework which supports the developer, there is less discussion on design space and minimum requirements for the target language for implementing such a system. The discussion about the design of DiaSuite is exclusively in the context of Java, and the implementation relies on a statically-typed object-oriented language. Our work explores the design space of compilers for declaration-driven frameworks, and maps out choices that can be made. Furthermore, we aim to provide programming language-independent principles to guide the design choices which will need to be made in future implement-

[2] Cassou, Balland et al. 2011; and Cassou, Bruneau et al. 2012

ations of such compilers and frameworks.

9.2 *Security of Android applications*

As mentioned in Chapter 2, there are a multitude of other approaches being investigated to provide safety to users of third-party applications on mobile platforms, notably on Android. That chapter already discusses general static analysis techniques, so we do not repeat those here.

The TouchDevelop approach by X. Xiao et al. 2012 is probably closest in spirit to our own, since they offer a methodology for developing applications using a simplified DSL (as opposed to writing the mobile application in a general-purpose programming language like Java). This approach makes static analysis feasible, since their language lacks many features which make analysis of a general-purpose language a hard problem, such as aliasing, hidden control flows, *etc.*[3] After a developer has submitted an application to the central repository for download by the user, they show the user which information flows an application might contain. Our approach, on the other hand, does not make use of static analysis techniques, but restricts information flow by construction. Furthermore, our methodology supports general-purpose programming languages, meaning that a developer is not forced to write applications in a restrictive language. Also, as a side-effect, existing libraries that a developer is accustomed to are still available. In addition, our approach offers the same level of insight for users, since arrows similar to those of TouchDevelop can be trivially derived from the specification our methodology is based on (see Definition 1 in Section 4.2).

[3] Atanas Rountev, Scott Kagan and Michael Gibas (2004). 'Evaluating the Imprecision of Static Analysis'. In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '04. Washington DC, USA: ACM, pp. 14–16.

Many other approaches have been developed. One example, specifically aimed at the privacy problem inherent in advertisement libraries is AdDroid.[4] It proposes privilege separation specifically for advertisement libraries by integrating them into the Android run-time library. While this approach is effective, we believe that our more general methodology will better accommodate future needs. Currently, advertising libraries are a major privacy threat,[5] but perhaps in the future another type of functionality will need to be integrated, and it does not seem scalable to update the run-time library every time new functionality is introduced. Our methodology separates all components, which we argue is a preferable approach. Other approaches include Apex,[6] which allows a user to specify constraints such as time and location for when specific permissions are to be allowed—for example, the application may only access the Internet during office hours. This approach could complement our own, and support could be built into the generated framework. For example, we might extend the declaration grammar to include such constraints.

[4] Paul Pearce et al. (2012). 'AdDroid: Privilege Separation for Applications and Advertisers in Android'. In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '12. Seoul, Korea: ACM, pp. 71–72.

[5] Stevens et al. 2012

[6] Mohammad Nauman, Sohail Khan and Xinwen Zhang (2010). 'Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints'. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '10. Beijing, China: ACM, pp. 328–332.

Finally, we point the reader to two comprehensive surveys of existing work that attempts to secure Android devices.[7] Another

[7] Sufatrio et al. (2015). 'Securing Android: A Survey, Taxonomy, and Challenges'. In: *ACM Comput. Surv.* 47.4, 58:1–58:45.; and Asim S. Yuksel, Abdul H. Zaim and Muhammed A. Aydin (2014). 'A Comprehensive Analysis of Android Security and Proposed Solutions'. In: *International Journal of Computer Network and Information Security* 12, pp. 9–20.

older publication surveys attacks on Android in a more general sense (that is, not just privacy leaks), and suggests avenues for further research.[8] These suggestions include finer-grained declarations, and an information-flow tracking system, as well as mitigations related to physically securing devices. However, we consider this outside the scope of the present work.

[8] Timothy Vidas, Daniel Votipka and Nicolas Christin (2011). 'All Your Droid Are Belong to Us: A Survey of Current Android Attacks'. In: *Proceedings of the 5th USENIX Conference on Offensive Technologies*. WOOT'11. San Francisco, CA: USENIX Association, pp. 10–10.

9.3 Operating system security

Another area of research which would complement our approach is capability systems. These operating systems aim to sandbox or isolate programs with restricted permissions.

Android builds on the Linux kernel, which implements the traditional coarse-grain user-based permission model, making it difficult to follow the principle of least authority (POLA).[9] In an attempt to achieve POLA, Android runs each application process as a separate user. However, over the years, Linux has been providing more fine-grained isolation mechanisms: this includes namespaces,[10] which form the basis of so-called isolated containers, as well as mechanisms to restrict the system calls available to user processes, such as seccomp or Mbox.[11]

Operating systems research has been focusing on *capability-based security*, including KeyKOS,[12] then EROS,[13] and finally Coyotos.[14] These approaches seem superior in that they offer true separation of privilege, provided by the operating system. This way, one can be sure that child processes do not escalate permissions compared to their parent processes. Capsicum provides yet another approach to restricting system calls via capabilities.[15] For a general overview of capability-based systems, see Levy 1984.

Using a capability-based system kernel approach would offer mobile platform (and other) users a much greater degree of safety regarding the use of their private resources. Unfortunately, this would require a very profound change to the software and infrastructure already widely deployed in mobile platforms. Capability-based operating systems are not currently widespread. Our methodology on the other hand requires only minimal modifications (if any) compared to the programming language tools and run-time support libraries already deployed, thus making its adoption more feasible.

[9] J. H. Saltzer and M. D. Schroeder (1975). 'The protection of information in computer systems'. In: *Proceedings of the IEEE* 63.9, pp. 1278–1308.

[10] Eric W. Biederman (2006). 'Multiple Instances of the Global Linux Namespaces'. In: *Proceedings of the Linux Symposium*. Vol. 1. Ottowa, Ontario, Canada, pp. 101–112

[11] The Linux Kernel Developers (2015). *SECure COMPuting with filters*. Online, `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`. Accessed: August 2015; and Taesoo Kim and Nickolai Zeldovich (2013). 'Practical and Effective Sandboxing for Non-root Users'. In: *USENIX Annual Technical Conference*. San Jose, CA: USENIX, pp. 139–144.

[12] Norman Hardy (1985). 'KeyKOS Architecture'. In: *SIGOPS Oper. Syst. Rev.* 19.4, pp. 8–25.

[13] Jonathan S. Shapiro, Jonathan M. Smith and David J. Farber (1999). 'EROS: A Fast Capability System'. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP '99. Charleston, South Carolina, USA: ACM, pp. 170–185.

[14] J. S. Shapiro et al. (2004). 'Towards a verified, general-purpose operating system kernel'. In: *Proceedings of the NICTA Formal Methods Workshop on Operating Systems Verification*. Ed. by G. Klein. NICTA Technical Report 0401005T-1. Sydney, Australia: National ICT Australia

[15] Robert N. M. Watson et al. (2010). 'Capsicum: Practical Capabilities for UNIX'. in: *Proceedings of the USENIX Security Symposium*. USENIX.

9.4 Language-level restrictions

Finally, we consider language-based approaches, which are similar to the present work. These attempt to define programming languages in such a way that it is possible to prevent access to arbitrary library code at the language level. One example is Mark Miller's *E* language,[16] and the accompanying run-time library called *ELib*. As a pure-Java library, *ELib* provides inter-process capability-secure distributed programming. Its cryptographic cap-

[16] Mark Samuel Miller (2006). 'Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control'. PhD thesis. Baltimore, Maryland, USA: Johns Hopkins University

ability protocol enables mutually suspicious Java processes to cooperate safely. Objects written in the *E* language are only able to interact with other objects according to the *ELib* semantics, enabling intra-process security with object-level granularity, including the ability to safely run untrusted library code. This technique could be used to complement our approach.

Another language-level approach is that offered by W7.[17] The W7 approach achieves a similar goal as the *E* language, but in a slightly modified variant of the Scheme programming language. Rees shows that the primitives of Scheme suffice to support secure sharing, authentication, and more, with security properties ensured by the Scheme implementation—the 'security kernel'. For example, protection is achieved using *closures*: a procedure is not just a program, but a program coupled with its environment of origin. A procedure cannot access the environment of its caller, and the caller cannot access the procedure's environment. The caller and callee are therefore protected from one another. Sharing is accomplished through explicitly shared portions of environments, which may include procedures that allow still other objects to be shared. This allows a much finer grain of control over inter-object communication than our Racket prototype allows by default. It is worth noting that Racket also supports sandboxed evaluation, with which similar control over data sharing should be feasible.

[17] Jonathan Allen Rees (1995). 'A security kernel based on the lambda-calculus'. PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology

« La rage de vouloir conclure est une des manies les plus funestes et les plus stériles qui appartiennent à l'humanité. Chaque religion et chaque philosophie a prétendu avoir Dieu à elle, toiser l'infini et connaître la recette du bonheur. Quel orgueil et quel néant ! Je vois, au contraire, que les plus grands génies et les plus grandes œuvres n'ont jamais conclu. »

Gustave Flaubert (1973). *Correspondance (1830–juin 1851)*. Ed. by Jean Bruneau. Vol. 1. Bibliothèque de la Pléiade. Paris, France: Éditions Gallimard, pp. 679–680 (Lettre du 4 septembre 1850 à Louis Bouilhet).

10
Conclusion and perspectives

We have presented a programming language-independent methodology to develop declaration-driven frameworks for open platforms, as defined in Section 2.3. Such platforms include diverse application domains such as home automation, assisted living, and mobile computing. Our methodology addresses concerns arising in these platforms, where end-users must provide sensitive information to be able to use third-party applications. We have shown that declaration-driven frameworks fill the need for user privacy and provide insight into application behaviour, which was previously not addressed by mainstream approaches. More fine-grained specifications provide the end-user with greater insight into the usage of their resources, in addition to being able to provide the programming framework and run-time library with richer structural and behavioural information to enable restrictions and guarantees.

Our methodology provides developer support throughout the development life-cycle of applications, starting with the design phase through to application implementation. Although not yet fully implemented, direction is given for a distribution model for applications, which maintains the high level of confidence that our methodology provides to the end-user.

This dissertation provides a specification for implementing instances of our framework generator methodology, from high-level semantics through to low-level implementation details, in real-world programming languages. We therefore expect our work to be able to serve as inspiration for future implementers of frameworks that support open platforms. The principles we provide in Chapter 7 should prevent pitfalls and help framework designers choose among the guarantees that can be provided, and how to implement them. Additionally, the discussion concerning the impact of the phase in which guards are placed should give concrete guidance to platform owners, when deciding how to control access to various shared resources.

Apart from the methodology, we have also provided two implementations of declaration compilers targeting significantly different programming languages. These implementations demonstrate that

it is feasible to implement the methodology, and that it offers tangible benefits in terms of developer support and user privacy. It also provides a guide for translating the high-level concepts of Section 4.3 into programming language constructs. Crucially, we have argued that the guarantees we are able to provide do not depend on the use of any particular programming language.

10.1 Assessment

We now assess our tool-based methodology with respect to the initial objectives defined in Section 1.1.

Behavioural transparency. Our approach demonstrates that by requiring developers to declare not only a list of permissions but also a high-level structural overview of the application, we can provide a high degree of predictability to the end-user of the application. By encouraging the decomposition of applications into contexts which are independently granted access to sensitive resources, the flow of data is made more explicit. This is an improvement over what was previously considered sufficient by massively adopted open platforms, such as Android or Apple's iOS. Instead of having tenuous insight into the usage of their private data, end-users can be provided with interfaces to inspect the component diagram of an application, and intuitively understand if and when there is a clear separation between a sensitive source, the components which query it, and public sinks.

Coherence between specification and behaviour. We have shown that without doing expensive static analysis on either the developer's code or the compiled binary, we can provide a high degree of confidence in the conformance between the specification of an application and how it behaves. This is possible since a generated framework can make it impossible for contexts to access arbitrary components, other than those to which access is explicitly granted. While further research is necessary to implement a run-time support library which addresses the limitations of our prototypes, we argue that by using techniques that are already accepted in mainstream frameworks, it is possible to enforce the properties arising from the declarations.

Development support. Apart from using the declarations provided by the developer to offer the end-user insight and control, they also allow the developer to benefit from the rich information that is provided about the application. Using this structural information, we are able to customise the development environment. We provide the developer with a focused API, and we reduce the need for repetitive development tasks implementing generic mechanisms, such as inter-component communication. This is all taken care of by the generated code. We hope our tools can improve the developer

experience and increase application quality. This should in turn drive adoption of the platform, making it more valuable.

Abstraction over host language. This methodology has been shown to be applicable to object-oriented and functional programming languages, with static or dynamic type systems. This suggests that the methodology is indeed language-independent. The only requirement is that the language offers a pre-runtime phase, but even that requirement is soft: an external declaration compiler can be implemented to fill that need.

10.2 Ongoing and future work

The work presented in this dissertation is being expanded in various directions. Here, we detail the possibilities for future work which builds on the material presented in the preceding chapters. The suggestions are sorted in order of increasing time scale and complexity: for example, the first one is a relatively straightforward modification, whereas the final suggestions would require serious research and time investment.

Parameterised specifications

The specification language used as an example throughout this dissertation was intentionally kept minimal, to allow us to focus on core semantics without having to deal with superfluous implementation details. However, a simple extension to the declaration language would make it much more expressive. Research should be done into the feasibility of providing a list of authorised parameter values to actions. For example, one might grant permission to only get access to the phone number of address book entries, or to get read-only access to specific files, or only allow a fixed list of URLs to be accessed. This will enable a user to authorise the application for execution, secure in the knowledge that only certified remote parties can be contacted.

In the generated framework this might be achieved by providing actions using a parameter with a constrained set of allowed values—*e.g.*, via a generated `enum` in Java. This way, the controller calling an action may only choose from a restricted set of values, or else a compile-time error is triggered.

Apply the methodology to assisted living

The research group is currently working towards applying the methodology to the domain of assisted living, where frail users have assistive applications to aid and monitor them. This is the stated purpose of the DomAssist project.[1] This raises privacy concerns, making our methodology a good candidate for application development for this scenario. Additionally, this validates our claim

[1] Loïc Caroux et al. (2014). 'Verification of Daily Activities of Older Adults: A Simple, Non-Intrusive, Low-Cost Approach'. In: *ASSETS–The 16th International ACM SIGACCESS Conference on Computers and Accessibility*. Rochester, NY, United States, pp. 43–50.

that the methodology is more widely applicable than just the domain of mobile computing.

Log-based development

Work is being done in the research group to collect data from sensors (movement sensors, energy usage sensors, *etc.*) installed in the homes of older adults for use in simulated testing of assisted living applications.[2] The realistic data from such installations will be useful for testing new assistive applications (examples might include monitoring that an older adult has completed a morning wake-up routine, or has not forgotten to lock the front door)[3] before going to the expensive effort of deploying the devices and software in a home. This can be supported by our methodology, since all resources are managed by the platform, making it easy to substitute devices for mock resources during the application testing phase.

[2] For more information about the HomeAssist research project referred to, see `http://phoenix.inria.fr/research-projects/homeassist`.

[3] These examples are taken from Caroux et al. 2014.

Using specifications to drive program analysis

Another promising direction is to guide program analysis tools based on architectural invariants. Our generative approach could automatically add architectural invariants as axioms to the model, facilitating verification. For example, we are investigating the verification of safety properties by injecting the architectural invariants from the Core DiaSpec specification in the model checker JPF.[4] Alternatively, generating flow constraints from the specification for use in a system inspired by iflowtypes.js[5] should be investigated. The iflowtypes.js library is designed to inline an information flow monitor into JavaScript code.

The project AppGuarden[6] is likely to be a good fit for this approach, since it is more specialised to the mobile computing domain than JPF. It proposes a tool called EviCheck, which is reported to be able to statically analyse an Android application for conformance to data flow rules, but to date little information is given. It seems very promising, however, to automatically generate such data flow rules from our declarations and have them analysed by EviCheck.[7]

[4] Willem Visser et al. (2003). 'Model Checking Programs'. In: *Automated Software Engineering* 10.2, pp. 203–232.

[5] Manuel Serrano et al. (2014). *Project-Team INDES–Secure Diffuse Programming*. Tech. rep. AR-2014-INDES. Sophia Antipolis, France: INRIA Sophia Antipolis–Méditerranée.

[6] Mobility + Security Group, University of Edinburgh (2015). *AppGuarden*. `http://groups.inf.ed.ac.uk/security/appguarden/Home.html`. Accessed: February 2015

[7] Joseph Hallett and David Aspinall (2014). 'Towards an authorization framework for app security checking'. In: *Engineering Secure Software and Systems (ESSoS) PhD Symposium*. Munich, Germany.

Improving run-time support

One of the limitations of our prototypes is the fact that a malicious developer can use unsupervised APIs, including system calls (such as writing to a file) to circumvent the separation of components. If a developer can communicate between components that should in fact be separated, privacy leaks are possible. Authorisation is ultimately handled by the operating system and run-time support library. While our compilers can enforce strict data flow in the generated code, this effort could easily be annihilated if the run-time support and OS are giving processes privileges that would allow them to effectively escape the data flow constraints.

This limitation only exists because our prototypes do not provide a customised run-time library. The capability-based operating systems approaches referenced in Section 9.3 illustrate that this is not a fundamental limitation. Therefore, a logical next step in pushing this research project forward is to provide a run-time library which restricts system calls. For example, the platform should prevent writing of files except via authorised routes that the user must approve (for example, a `Filesystem` action and source). This would be possible by modifying the JVM, or in the case of Racket, providing no access to any base library functions. This should be feasible using the sandboxed evaluation features of Racket.[8]

[8] Matthew Flatt and PLT (2015b). 'Sandboxed Evaluation'. In: PLT-TR-2010-1. http://racket-lang.org/tr1/, Version 6.2.1. Chap. 14.12.

Design an application store

In this dissertation, hints are given towards a viable distribution model of applications developed using our methodology. However, more research into a practical design for an application store supporting developers and users of applications is required. Currently, we assume that the developer will upload the specification and the source code of their application to a server controlled by the platform owner, where the software will be compiled and packaged, ready for download and installation by an end-user. However, because of intellectual property concerns this is not a realistic model—a developer cannot always be expected to upload source code.

Android and Apple's iOS, for example, are supported by application stores which accept application bundles as binaries. We expect that our methodology will still provide the same guarantees if the developer is allowed to upload binaries of their application components. For example, the developer could be asked to upload the Java bytecode class files, potentially packaged as OSGi containers with contracts,[9] for their components, which would be combined with the framework classes generated on the platform owner's server. However, further research is required to preclude the possibility that malicious developers could subvert the compiled application to perform unauthorised operations.

[9] Alexandre de Castro Alves (2011). *OSGi in Depth*. 1st edition. Greenwich, CT, USA: Manning Publications Co.

User acceptability study

This work has focused on the fundamentals behind a user privacy framework. However, we have not considered whether users will be able to understand the implications of the type of information that we provide—*e.g.*, arrows from sources to sinks, or the application diagram with components. While there is research towards user comprehension of the privacy implications of permissions,[10] it would be interesting to study whether our approach improves on the status quo as far as presentation of data flow goes, or not.

[10] Felt et al. 2012

We remark that if it turns out that users prefer the way Android currently displays permissions, we can easily retrieve any level

of granularity desired from our style of specifications. We can, in fact, offer increased information flow control, while not necessarily changing the current user interface. Further research involving a user study would provide more information for directions of development.

Verified implementation

Currently our methodology has a semantic gap: declarations are interpreted separately by a compiler, which then outputs programming infrastructure (the framework) which enforces certain properties (containment of data, or module separation). This translation from declarations into programming artefacts is not verified, and relies on a common sense evaluation and manual testing of the resulting framework code. It would likely be a very ambitious project, but implementing our methodology entirely (that is, from specification translation through to hosting the final application) inside a dependently-typed language or proof assistant such as Agda, Idris, or Coq[11] would be a very interesting experiment. In the best case, it might be possible to prove properties on a given specification implemented as a data structure in the dependently-typed language, and write translation functions which can be proven to preserve those properties. It would be interesting to prove key parts of the declaration compiler, following the lead of work such as the CompCert verified C compiler[12] and the seL4 verified general-purpose microkernel.[13]

[11] Ulf Norell (2009). 'Dependently Typed Programming in Agda'. In: *Advanced Functional Programming*. Ed. by Pieter Koopman, Rinus Plasmeijer and Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 230–266.; Edwin C. Brady (2011). 'IDRIS—Systems Programming Meets Full Dependent Types'. In: *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*. PLPV '11. Austin, Texas, USA: ACM, pp. 43–54.; and The Coq development team (2004). *The Coq proof assistant reference manual*. Version 8.0. LogiCal Project.

[12] Xavier Leroy (2014). *The CompCert C verified compiler: Documentation and user's manual*. Tech. rep. Inria.

[13] T. Murray et al. (2013). 'seL4: From General Purpose to a Proof of Information Flow Enforcement'. In: *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 415–429.

Appendices

A
Code listings

A.1 Racket code listings

Core DiaSpec language grammar

Figure 27 presents the grammar used in the PLT Redex model of Core DiaSpec. There are minor differences with the grammar used in the prototypes. For example, the terms are parenthesised in the style of Scheme, and the use of (get nothing) to indicate the absence of a data requirement is mandatory. This grammar is nevertheless completely equivalent to the grammar presented in Figure 4 of Chapter 3.

This grammar also includes typing terms, used in the type checking phase of the compiler. See Section 4.2.

```
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

            Γ ::= ((X : t) ...)
            t ::= (ACT τ)
                | (SRC τ)
                | (CTX-req τ)
                | (CTX-prov τ)
                | (CTRL)
```

Figure 27: Complete grammar of the Core DiaSpec specification language as modelled in PLT Redex, extended with type environments.

Complete Core DiaSpec type system

For completeness we present the full list of typing rules from Section 4.2 here. Note that the following figure spans multiple pages. The metafunctions used in the type judgements are presented in the next figure, Figure 29.

$$\frac{\text{unique?}[\![X, \Gamma]\!]}{\vdash [\![\Gamma, (\text{source } X \text{ as } \tau), (\text{SRC } \tau)]\!]}\ \text{[intro-src]}$$

$$\frac{\text{unique?}[\![X, \Gamma]\!]}{\vdash [\![\Gamma, (\text{action } X \text{ as } \tau), (\text{ACT } \tau)]\!]}\ \text{[intro-act]}$$

$$\frac{\text{unique?}[\![X, \Gamma]\!]}{\vdash [\![\Gamma, (\text{context } X \text{ as } \tau \text{ [when required (get nothing)]}), (\text{CTX-req } \tau)]\!]}\ \text{[ctx-req-get-ø]}$$

$$\frac{(\text{SRC } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \quad \text{unique?}[\![X_1, \Gamma]\!]}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when required (get } X_2\text{)]}), (\text{CTX-req } \tau_1)]\!]}\ \text{[ctx-req-get-src]}$$

$$\frac{(\text{CTX-req } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \quad \text{unique?}[\![X_1, \Gamma]\!]}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when required (get } X_2\text{)]}), (\text{CTX-req } \tau_1)]\!]}\ \text{[ctx-req-get-ctx]}$$

$$\frac{(\text{SRC } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \quad \text{unique?}[\![X_1, \Gamma]\!]}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when provided } X_2 \text{ (get nothing) _]}), (\text{CTX-prov } \tau_1)]\!]}\ \text{[ctx-onSrc-get-ø]}$$

$$\frac{(\text{CTX-prov } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \quad \text{unique?}[\![X_1, \Gamma]\!]}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when provided } X_2 \text{ (get nothing) _]}), (\text{CTX-prov } \tau_1)]\!]}\ \text{[ctx-onCtx-get-ø]}$$

Figure 28: The complete list of type judgements for Core DiaSpec specifications.

$$\frac{\begin{array}{c}(\text{SRC } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \\ (\text{SRC } \tau_3) = \text{lookup}[\![\Gamma, X_3]\!] \\ \text{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when provided } X_2 \text{ (get } X_3\text{) _])}, (\text{CTX-prov } \tau_1)]\!]}\ \text{[ctx-onSrc-get-src]}$$

$$\frac{\begin{array}{c}(\text{CTX-prov } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \\ (\text{SRC } \tau_3) = \text{lookup}[\![\Gamma, X_3]\!] \\ \text{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when provided } X_2 \text{ (get } X_3\text{) _])}, (\text{CTX-prov } \tau_1)]\!]}\ \text{[ctx-onCtx-get-src]}$$

$$\frac{\begin{array}{c}(\text{SRC } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \\ (\text{CTX-req } \tau_3) = \text{lookup}[\![\Gamma, X_3]\!] \\ \text{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when provided } X_2 \text{ (get } X_3\text{) _])}, (\text{CTX-prov } \tau_1)]\!]}\ \text{[ctx-onSrc-get-ctx]}$$

$$\frac{\begin{array}{c}(\text{CTX-prov } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \\ (\text{CTX-req } \tau_3) = \text{lookup}[\![\Gamma, X_3]\!] \\ \text{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\text{context } X_1 \text{ as } \tau_1 \text{ [when provided } X_2 \text{ (get } X_3\text{) _])}, (\text{CTX-prov } \tau_1)]\!]}\ \text{[ctx-onCtx-get-ctx]}$$

$$\frac{\begin{array}{c}(\text{CTX-prov } \tau_2) = \text{lookup}[\![\Gamma, X_2]\!] \\ (\text{ACT } \tau_3) = \text{lookup}[\![\Gamma, X_3]\!] \\ \text{unique?}[\![X_1, \Gamma]\!]\end{array}}{\vdash [\![\Gamma, (\text{controller } X_1 \text{ [when provided } X_2 \text{ do } X_3\text{])}, (\text{CTRL})]\!]}\ \text{[intro-controller]}$$

$$\frac{\begin{array}{c}\vdash [\![\Gamma, \textit{declaration}, t]\!] \\ X_{new} = \text{varname}[\![\textit{declaration}]\!]\end{array}}{\text{decl-ok}[\![\Gamma, \textit{declaration}, (X_{new} : t)]\!]}\ \text{[decl-ok]}$$

$$\frac{\text{check-spec}[\![\Gamma, \textit{specification}, \Gamma_2]\!]}{\text{spec-ok}[\![\Gamma, \textit{specification}]\!]}\ \text{[type-erasure]}$$

$$\frac{}{\text{check-spec}[\![\Gamma_1, (), \Gamma_1]\!]}\ \text{[empty-spec]}$$

$$\frac{\begin{array}{c}(\textit{declaration}_1\ \textit{declaration}_2\ ...) = \textit{specification} \\ \text{decl-ok}[\![\Gamma, \textit{declaration}_1, (X : t)]\!] \\ \text{check-spec}[\![\text{extend}[\![\Gamma, (X : t)]\!], (\textit{declaration}_2\ ...), \Gamma_2]\!]\end{array}}{\text{check-spec}[\![\Gamma, \textit{specification}, \Gamma_2]\!]}\ \text{[check-spec]}$$

Type system metafunctions

The metafunction varname simply returns the binder name of a component declaration. The lookup function finds the type of a binder in the type environment, if it exists. The metafunction extend takes an environment and one or more pairs associating a variable with a type, and returns the environment with the new pairs prepended. Finally, unique? checks that a given variable does not yet appear in a type environment.

varname : *declaration* → X
varname⟦(source X as any_1)⟧ = X
varname⟦(action X as any_1)⟧ = X
varname⟦(context X as any_1 any_2)⟧ = X
varname⟦(controller X any_2)⟧ = X
lookup : Γ X → t or #f
lookup⟦((X_1 : t) (X_2 : t_2) ...), X_1⟧ = t
lookup⟦((X_1 : t) (X_2 : t_2) ...), X_3⟧ = lookup⟦((X_2 : t_2) ...), X_3⟧
lookup⟦(), X⟧ = #f
extend : Γ (X : t) ... → Γ
extend⟦((X_Γ : t_Γ) ...), (X : t), ...⟧ = ((X : t) ... (X_Γ : t_Γ) ...)
unique? : X Γ → *boolean*
unique?⟦X, ((X_1 : _) ... (X : _) (X_2 : _) ...)⟧ = #f
unique?⟦X, *any*⟧ = #t

Figure 29: Definitions of the metafunctions used in the type judgements.

A.2 Java code listings

Option type, `Maybe<T>`

```
1  abstract public class Maybe<T> { }
2
3  public class Just<T> extends Maybe<T> {
4
5    public T just_value;
6
7    public Just (T v){
8      this.just_value = v;
9    }
10 }
11
12 public class Nothing<T> extends Maybe<T> { }
```

Figure 30: The option type, `Maybe<T>`. Implemented as 3 separate classes.

A.3 Racket code listings

Helper macros

```
(define-syntax (implement sstx)
  (syntax-case sstx []
    [(_ name as (... ...))
     (with-syntax
       ([ins (make-id "implement-~a" sstx  #'name)])
       ;; check that x is declared
       (unless (ormap (lambda (x) (equal? x (syntax->datum #'name)))
                      (append (storage-now rest) (storage-now taxo)))
         (raise-syntax-error
          ;; dev called (implement x ..) where x wasn't declared in spec
          (syntax->datum #'name) " is not defined in " #,(mymodname)))
       #'(begin (ins as (... ...))))]))
```

Figure 31: Helper macro to translate terms of the form `(implement x ...)` into `(implement-x ...)`.

```
(provide (rename-out
          [implementation-module-begin #%module-begin]))

(define-syntax (implementation-module-begin stx2)
  (syntax-case stx2 (implement taxonomy)
    [(_ (taxonomy file)
        (implement decls (... ...)) (... ...))
     ;; splice in the taxonomy implementations
     (with-syntax ([(taxo (... ...))
                    (datum->syntax stx2 (port->syntax
                                         (open-input-file
                                          (syntax->datum #'file)) (list)))])
       ;; make sure all declared components are implemented
       (check-presence-of-implementations
        (storage-now rest) #'((implement decls (... ...)) (... ...)))

       #'(#%module-begin
          (require "memory.rkt")
          (emptyHash)
          taxo (... ...)    ; include syntax from taxo-file
          (implement decls (... ...)) (... ...)))]))
```

Figure 32: Snippet spliced into all specification modules at expansion time. The `#%module-begin` macro allows the specification file to be used as a Racket language extension.

Bibliography

The entries in the bibliography are all sorted by the last name of the primary author. Dashes signify that the list of authors is the same as the preceding bibliographic entry.

Yuvraj Agarwal and Malcolm Hall (2013). 'ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing'. In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '13. Taipei, Taiwan: ACM, pp. 97–110. ISBN: 9781450316729. URL: http://doi.acm.org/10.1145/2462456.2464460.

Emilie Balland and Charles Consel (2010). 'Open Platforms: New Challenges for Software Engineering'. In: *Programming Support Innovations for Emerging Distributed Applications*. PSI EtA '10. Reno, Nevada: ACM, 3:1–3:4. ISBN: 9781450305440. URL: http://doi.acm.org/10.1145/1940747.1940750.

Alexandre Bartel, Jacques Klein, Yves Le Traon and Martin Monperrus (2012). 'Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android'. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, pp. 274–277. ISBN: 9781450312042. URL: http://doi.acm.org/10.1145/2351676.2351722.

Eric W. Biederman (2006). 'Multiple Instances of the Global Linux Namespaces'. In: *Proceedings of the Linux Symposium*. Vol. 1. Ottowa, Ontario, Canada, pp. 101–112.

Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales and David A. Moon (1988). 'Common Lisp Object System Specification'. In: *SIGPLAN Not.* 23.SI, pp. 1–142. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/885631.885632.

Edwin C. Brady (2011). 'IDRIS—Systems Programming Meets Full Dependent Types'. In: *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification*. PLPV '11. Austin, Texas, USA: ACM, pp. 43–54. ISBN: 9781450304870. DOI: 10.1145/1929529.1929536.

Julien Bruneau and Charles Consel (2013). 'DiaSim: a simulator for pervasive computing applications'. In: *Software: Practice and Experience* 43.8, pp. 885–909. DOI: 10.1002/spe.2130.

Ed Burnette (2009). *Hello, Android: Introducing Google's Mobile Development Platform*. 2nd edition. Pragmatic Bookshelf. ISBN: 1934356492, 9781934356494.

Nicholas Cameron and James Noble (2010). 'Encoding ownership types in Java'. In: *Objects, Models, Components, Patterns*. Springer, pp. 271–290.

Loïc Caroux, Charles Consel, Lucile Dupuy and Hélène Sauzéon (2014). 'Verification of Daily Activities of Older Adults: A Simple, Non-Intrusive, Low-Cost Approach'. In: *ASSETS–The 16th International ACM SIGACCESS Conference on Computers and Accessibility*. Rochester, NY, United States, pp. 43–50. URL: https://hal.inria.fr/hal-01015280.

Damien Cassou (2011). 'Développement logiciel orienté paradigme de conception : la programmation dirigée par la spécification'. PhD thesis. Université Sciences et Technologies–Bordeaux I. URL: https://tel.archives-ouvertes.fr/tel-00583246.

Damien Cassou, Emilie Balland, Charles Consel and Julia Lawall (2011). '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. Waikiki, Honolulu, HI, USA: ACM, pp. 431–440. ISBN: 9781450304450. URL: http://doi.acm.org/10.1145/1985793.1985852.

Damien Cassou, Julien Bruneau, Charles Consel and Emilie Balland (2012). 'Toward a Tool-Based Development Methodology for Pervasive Computing Applications'. In: *IEEE Trans. Software Eng.* 38.6, pp. 1445–1463.

Alexandre de Castro Alves (2011). *OSGi in Depth*. 1st edition. Greenwich, CT, USA: Manning Publications Co. ISBN: 193518217X, 9781935182177.

Chrome developers (2015). *Developing Chrome Extensions: Declare Permissions*. https://developer.chrome.com/extensions/declare_permissions. Accessed: February 2015.

Gregory H. Cooper and Shriram Krishnamurthi (2004). *FrTime: Functional reactive programming in PLT Scheme*. Tech. rep. CS-03-20. Providence, Rhode Island, USA: Brown University.

Antony Courtney (2001). 'Frappé: Functional Reactive Programming in Java'. In: *Practical Aspects of Declarative Languages*. Ed. by I. V. Ramakrishnan. Vol. 1990. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 29–44. ISBN: 9783540417682. DOI: 10.1007/3-540-45241-9_3.

Matthias Kalle Dalheimer (2010). *Programming with QT: Writing portable GUI applications on Unix and Win32*. O'Reilly Media.

Luis Damas and Robin Milner (1982). 'Principal Type-schemes for Functional Programs'. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: ACM, pp. 207–212. ISBN: 0897910656. URL: http://doi.acm.org/10.1145/582153.582176.

Pierre Deransart, Martin Jourdan and Bernard Lorho (1988). *Attribute grammars: definitions, systems, and bibliography*. Lecture notes in computer science. Berlin, New York: Springer-Verlag. ISBN: 0387500561. URL: http://opac.inria.fr/record=b1087092.

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan and Matthias Felleisen (2011). 'Correct blame for contracts: no more scapegoating'. In: *ACM SIGPLAN Notices*. Vol. 46. ACM, pp. 215–226.

Quang Do, Ben Martini and Kim-Kwang Raymond Choo (2015). 'Exfiltrating data from Android devices'. In: *Computers & Security* 48, pp. 74–91. ISSN: 0167-4048. URL: http://www.sciencedirect.com/science/article/pii/S016740481400162X.

Lucile Dupuy, Hélène Sauzéon and Charles Consel (2015). 'Perceived Needs for Assistive Technologies in older adults and their caregivers'. In: *womENcourage 2015*. Uppsala, Sweden. URL: https://hal.archives-ouvertes.fr/hal-01168399.

Karim O. Elish, Danfeng Daphne Yao, Barbara G. Ryder and Xuxian Jiang (2013). 'A static assurance analysis of Android applications'. In: *Virginia Polytechnic Institute and State University, Tech. Rep*.

Quentin Enard (2013). 'Development of dependable applications: a design-driven approach'. PhD thesis. Université Sciences et Technologies–Bordeaux I. URL: https://tel.archives-ouvertes.fr/tel-00829477.

William Enck (2011). 'Defending Users against Smartphone Apps: Techniques and Future Directions'. In: *Information Systems Security*. Ed. by Sushil Jajodia and Chandan Mazumdar. Vol. 7093. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 49–70. ISBN: 9783642255595. URL: http://dx.doi.org/10.1007/978-3-642-25560-1_3.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel and Anmol N. Sheth (2014). 'TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones'. In: *Communications of the ACM* 57.3, pp. 99–106.

Mohamed Fayad and Douglas C. Schmidt (1997). 'Object-oriented application frameworks'. In: *Commun. ACM* 40.10, pp. 32–38. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/262793.262798.

Marc Feeley (2003). 'SRFI 39: Parameter objects'. In: *Scheme Requests for Implementation*. Ed. by Arthur A. Gleckler. http://srfi.schemers.org/srfi-39/srfi-39.html. Accessed: September 2015. Published online.

Jesse Feiler (2008). *How to Do Everything: Facebook Applications*. 1st edition. New York, NY, USA: McGraw-Hill, Inc. ISBN: 0071549676, 9780071549677.

Matthias Felleisen, Robert Bruce Findler and Matthew Flatt (2009). *Semantics Engineering with PLT Redex*. 1st edition. The MIT Press. ISBN: 0262062755, 9780262062756.

Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin and David Wagner (2012). 'Android permissions: User attention, comprehension, and behavior'. In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, p. 3. URL: http://dl.acm.org/citation.cfm?id=2335360.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler and Matthias Felleisen (2002). 'DrScheme: a programming environment for Scheme'. In: *J. Funct. Program.* 12.2, pp. 159–182. URL: http://dx.doi.org/10.1017/S0956796801004208.

Matthew Flatt (2013). 'Submodules in Racket: You Want It when, Again?' In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. GPCE '13. Indianapolis, Indiana, USA: ACM, pp. 13–22. ISBN: 9781450323734. URL: http://doi.acm.org/10.1145/2517208.2517211.

Matthew Flatt and PLT (2010). *Reference: Racket*. Tech. rep. PLT-TR-2010-1. http://racket-lang.org/tr1/, Version 6.2.1. PLT Design Inc.

– (2015a). 'Pattern-based Syntax Matching'. In: PLT-TR-2010-1. http://racket-lang.org/tr1/, Version 6.2.1. Chap. 12.1. URL: http://docs.racket-lang.org/reference/stx-patterns.html.

– (2015b). 'Sandboxed Evaluation'. In: PLT-TR-2010-1. http://racket-lang.org/tr1/, Version 6.2.1. Chap. 14.12. URL: http://docs.racket-lang.org/reference/Sandboxed_Evaluation.html.

Marcus Fontoura, Christiano Braga, Leonardo Moura and Carlos Lucena (2000). 'Using domain specific languages to instantiate object-oriented frameworks'. In: *IEE Proceedings–Software* 147.4, pp. 109–116.

Martin Fowler (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional.

– (2010). *Domain-specific languages*. Pearson Education.

Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau and Patrick McDaniel (2013). *Highly Precise Taint Analysis for Android Applications*. Tech. rep. TUD-CS-2013-0113. EC SPRIDE.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st edition. Addison-Wesley Professional. ISBN: 9780201633610.

Stéphanie Gatti (2014). 'A step-wise approach for integrating QoS throughout software development process'. PhD thesis. Université de Bordeaux. URL: https://hal.inria.fr/tel-01111456.

Clint Gibler, Jonathan Crussel, Jeremy Erickson and Hao Chen (2011). *AndroidLeaks: Detecting Privacy Leaks in Android Applications*. Tech. rep. UC Davis.

Joseph Hallett and David Aspinall (2014). 'Towards an authorization framework for app security checking'. In: *Engineering Secure Software and Systems (ESSoS) PhD Symposium*. Munich, Germany. URL: https://github.com/bogwonch/ESSoS2014/blob/master/paper/paper.pdf.

Norman Hardy (1985). 'KeyKOS Architecture'. In: *SIGOPS Oper. Syst. Rev.* 19.4, pp. 8–25. ISSN: 0163-5980. DOI: 10.1145/858336.858337.

S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, Feng Shen, S. Y. Ko and L. Ziarek (2013). 'Flow Permissions for Android'. In: *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pp. 652–657. DOI: 10.1109/ASE.2013.6693128.

R. Nigel Horspool and Nikolai Tillmann (2013). *TouchDevelop: Programming on the Go*. 3rd edition. The Expert's Voice. available at https://www.touchdevelop.com/docs/book. Apress. ISBN: 9781430261360.

Ken Hyland (2003). 'Dissertation Acknowledgements: The Anatomy of a Cinderella Genre'. In: *Written Communication* 20.3, pp. 242–268. DOI: 10.1177/0741088303257276. URL: http://wcx.sagepub.com/content/20/3/242.full.pdf+html.

Bill Joy, Guy Steele, James Gosling and Gilad Bracha (2000). *Java™ Language Specification*. Addisson-Wesley.

Taesoo Kim and Nickolai Zeldovich (2013). 'Practical and Effective Sandboxing for Non-root Users'. In: *USENIX Annual Technical Conference*. San Jose, CA: USENIX, pp. 139–144. ISBN: 9781931971010. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen and Bruce Duba (1986). 'Hygienic macro expansion'. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, pp. 151–161.

Xavier Leroy (2014). *The CompCert C verified compiler: Documentation and user's manual*. Tech. rep. Inria. URL: https://hal.inria.fr/hal-01091802.

Henry M. Levy (1984). *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann. ISBN: 0932376223.

Christopher Mann and Artem Starostin (2012). 'A framework for static detection of privacy leaks in Android applications'. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, pp. 1457–1462.

William C. Mann (2005). *Smart Technology for Aging, Disability, and Independence: The State of the Science*. 1st edition. Hoboken, NJ, USA: John Wiley and Sons. ISBN: 9780471696940.

Dave Mark and Jeff LaMarche (2009). *Beginning iPhone Development: Exploring the iPhone SDK*. Apress.

Mike McShaffry and David Graham (2012). *Game Coding Complete*. 4th edition. Independence, KY, USA: Cengage Learning PTR. ISBN: 9781133776574.

Hidden Android feature allows users to fine tune app permissions (2013). Online, http://www.zdnet.com/hidden-android-feature-allows-users-to-fine-tune-app-permissions-7000018944/. Accessed: May 2015.

Mark Samuel Miller (2006). 'Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control'. PhD thesis. Baltimore, Maryland, USA: Johns Hopkins University.

Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani and Riyadh Mahmood (2012). 'Testing Android Apps Through Symbolic Execution'. In: *SIGSOFT Softw. Eng. Notes* 37.6, pp. 1–5. ISSN: 0163-5948. URL: http://doi.acm.org/10.1145/2382756.2382798.

Mobility + Security Group, University of Edinburgh (2015). *AppGuarden*. http://groups.inf.ed.ac.uk/security/appguarden/Home.html. Accessed: February 2015.

Gilles Muller, Charles Consel, Renaud Marlet, L. P. Barreto, Fabrice Mérillon and Laurent Réveillère (2000). 'Towards robust OSes for appliances: A new approach based on Domain-Specific Languages'. In: *ACM SIGOPS European Workshop*. France, pp. 19–24. URL: https://hal.archives-ouvertes.fr/hal-00350228.

T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, Xin Gao and G. Klein (2013). 'seL4: From General Purpose to a Proof of Information Flow Enforcement'. In: *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 415–429. DOI: 10.1109/SP.2013.35.

Mohammad Nauman, Sohail Khan and Xinwen Zhang (2010). 'Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints'. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '10. Beijing, China: ACM, pp. 328–332. ISBN: 9781605589367. DOI: 10.1145/1755688.1755732.

Flemming Nielson, Hanne R. Nielson and Chris Hankin (1999). *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 3540654100.

Ulf Norell (2009). 'Dependently Typed Programming in Agda'. In: *Advanced Functional Programming*. Ed. by Pieter Koopman, Rinus Plasmeijer and Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 230–266. ISBN: 9783642046513. DOI: 10.1007/978-3-642-04652-0_5.

ORACLE (2015). *Java documentation: Annotations*. URL: http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html.

Paul Pearce, Adrienne Porter Felt, Gabriel Nunez and David Wagner (2012). 'AdDroid: Privilege Separation for Applications and Advertisers in Android'. In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '12. Seoul, Korea: ACM, pp. 71–72. ISBN: 9781450316484. DOI: 10.1145/2414456.2414498.

Benjamin Pierce (2002). *Types and programming languages*. Cambridge, MA, USA: MIT Press. ISBN: 0262162091.

Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis and Herbert Bos (2010). 'Paranoid Android: Versatile Protection for Smartphones'. In: *Proceedings of the 26th Annual Computer Security*

Applications Conference. ACSAC '10. Austin, Texas, USA: ACM, pp. 347–356. ISBN: 9781450301336. DOI: 10.1145/1920261.1920313.

Ruben Prieto-Diaz and Peter Freeman (1987). 'Classifying software for reusability'. In: *Software, IEEE* 4.1, pp. 6–16.

Jonathan Allen Rees (1995). 'A security kernel based on the lambda-calculus'. PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology.

Don Roberts and Ralph Johnson (1996). 'Evolve frameworks into domain-specific languages'. In: *3rd International Conference on Pattern Languages for Programming*. Monticelli, IL, USA.

Rick Rogers, John Lombardo, Zigurd Mednieks and Blake Meike (2009). *Android Application Development: Programming with the Google SDK*. Beijing, China: O'Reilly. ISBN: 9780596521479.

Atanas Rountev, Scott Kagan and Michael Gibas (2004). 'Evaluating the Imprecision of Static Analysis'. In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '04. Washington DC, USA: ACM, pp. 14–16. ISBN: 1581139101. DOI: 10.1145/996821.996829.

J. H. Saltzer and M. D. Schroeder (1975). 'The protection of information in computer systems'. In: *Proceedings of the IEEE* 63.9, pp. 1278–1308. ISSN: 0018-9219. DOI: 10.1109/PROC.1975.9939.

David A. Schmidt (1986). *Denotational Semantics: A Methodology for Language Development*. Dubuque, IA, USA: William C. Brown Publishers. ISBN: 0-697-06849-2.

Manuel Serrano, Nataliia Bielova, Ilaria Castellani, Tamara Rezk and Bernard Serpette (2014). *Project-Team INDES–Secure Diffuse Programming*. Tech. rep. AR-2014-INDES. Sophia Antipolis, France: INRIA Sophia Antipolis–Méditerranée. URL: https://raweb.inria.fr/rapportsactivite/RA2014/indes/uid14.html.

J. S. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar and M. Miller (2004). 'Towards a verified, general-purpose operating system kernel'. In: *Proceedings of the NICTA Formal Methods Workshop on Operating Systems Verification*. Ed. by G. Klein. NICTA Technical Report 0401005T-1. Sydney, Australia: National ICT Australia.

Jonathan S. Shapiro, Jonathan M. Smith and David J. Farber (1999). 'EROS: A Fast Capability System'. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP '99. Charleston, South Carolina, USA: ACM, pp. 170–185. ISBN: 1581131402. DOI: 10.1145/319151.319163.

Jeremy G. Siek and Walid Taha (2006). 'Gradual Typing for Functional Languages'. In: *Scheme and Functional Programming Workshop*, pp. 81–92.

Michael Snoyman (2012). *Developing Web Applications with Haskell and Yesod*. O'Reilly. ISBN: 9781449316976.

Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson and Hao Chen (2012). 'Investigating User Privacy in Android Ad Libraries'. In: *Workshop on Mobile Security Technologies (MoST)*.

Sufatrio, Darell J. J. Tan, Tong-Wei Chua and Vrizlynn L. L. Thing (2015). 'Securing Android: A Survey, Taxonomy, and Challenges'. In: *ACM Comput. Surv.* 47.4, 58:1–58:45. ISSN: 0360-0300. DOI: 10.1145/2733306.

S. Doaitse Swierstra, Pablo R. Azero Alcocer and João Saraiva (1999). 'Designing and Implementing Combinator Languages'. In: *Advanced Functional Programming*. Ed. by S. Doaitse Swierstra, José N. Oliveira and Pedro R. Henriques. Vol. 1608. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 150–206. ISBN: 9783540662419. URL: http://dx.doi.org/10.1007/10704973_4.

Richard N. Taylor, Nenad Medvidovic and Eric M. Dashofy (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing.

The Coq development team (2004). *The Coq proof assistant reference manual*. Version 8.0. LogiCal Project. URL: http://coq.inria.fr.

The Linux Kernel Developers (2015). *SECure COMPuting with filters*. Online, https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt. Accessed: August 2015.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt and Matthias Felleisen (2011). 'Languages as libraries'. In: *ACM SIGPLAN Notices*. Vol. 46. ACM, pp. 132–141.

Sam Tobin-Hochstadt and Matthew Flatt (2007). 'Advanced macrology and the implementation of Typed Scheme'. In: *In Proc. 8th Workshop on Scheme and Functional Programming*. ACM Press, pp. 1–14.

Timothy Vidas, Daniel Votipka and Nicolas Christin (2011). 'All Your Droid Are Belong to Us: A Survey of Current Android Attacks'. In: *Proceedings of the 5th USENIX Conference on Offensive Technologies*. WOOT'11. San Francisco, CA: USENIX Association, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=2028052.2028062.

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park and Flavio Lerda (2003). 'Model Checking Programs'. In: *Automated Software Engineering* 10.2, pp. 203–232. ISSN: 0928-8910. URL: http://link.springer.com/article/10.1023/A:1022920129859.

Robert N. M. Watson, Jonathan Anderson, Ben Laurie and Kris Kennaway (2010). 'Capsicum: Practical Capabilities for UNIX'. In: *Proceedings of the USENIX Security Symposium*. USENIX. URL: https://www.usenix.org/event/sec10/tech/full_papers/Watson.pdf.

Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu and Michalis Faloutsos (2012). 'Permission Evolution in the Android Ecosystem'. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida, USA: ACM, pp. 31–40. ISBN: 9781450313124. DOI: 10.1145/2420950.2420956.

Claud Xiao (2015). *Novel malware XcodeGhost modifies Xcode, infects Apple iOS apps and hits App Store*. Online. Accessed 29/9/2015. URL: http://researchcenter.paloaltonetworks.com/2015/09/

novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/.

Xusheng Xiao, Nikolai Tillmann, Manuel Fähndrich, Jonathan de Halleux and Michal Moskal (2012). 'User-aware privacy control via extended static information-flow analysis'. In: *ASE*. Ed. by Michael Goedicke, Tim Menzies and Motoshi Saeki. ACM, pp. 80–89. ISBN: 9781450312042.

Asim S. Yuksel, Abdul H. Zaim and Muhammed A. Aydin (2014). 'A Comprehensive Analysis of Android Security and Proposed Solutions'. In: *International Journal of Computer Network and Information Security* 12, pp. 9–20. DOI: 10.5815/ijcnis.2014.12.02.