Abstract—Today’s computationally able mobile devices are capable of acting as service providers as opposed to their traditional role as consumers. To address the challenges associated with the development of these mobile services, we have developed Odin, a middleware which masks complexity, allowing rapid development of mobile services. Odin, however, does not allow cross-platform development, which is an important concern with today’s wide variety of mobile devices. To solve this problem, we have designed OdinTools - a model-driven toolkit for cross-platform development of mobile services. Leveraging appropriate metamodels, a prototype has been implemented in Eclipse and Marama that allows developers to model mobile services in a platform-independent manner. We are currently working on transformations between levels of the model hierarchy which will allow full Odin-based service implementations to be generated automatically.

I. INTRODUCTION

Historically, mobile devices such as phones and PDAs have acted as consumers of services hosted by a fixed infrastructure (e.g. a 3G phone may act as a web browser). Today’s mid-range mobile devices are significantly more computationally able than their predecessors and are equipped with features such as GPS and the ability to connect to multiple networks such as 3G, Wi-Fi, and Bluetooth. Such advances enable mobile services, where mobile devices act as service providers, rather than their traditional role as consumers. Use of mobile devices as service providers allows mobile-specific features, for example location awareness and the means to communicate with auxiliary devices, to be leveraged in offering novel services.

The potential of mobile services can be illustrated with a collaborative health scenario. Consider a team of firefighters, each equipped with a body area network of sensors to monitor vital signs; and a mobile device exposing that information and the firefighter’s location, as a service. Team members can keep track of each other’s health and whereabouts using this system. Furthermore, if a member of the team encounters trouble during a rescue operation, this will be detected by the vital sign monitor. The operation coordinator, acting as a mobile service client, will see this and can then direct team members to provide assistance.

To create services such as these, developers have traditionally had to overcome many difficulties associated with mobile development. These include limited computational resources, limited and intermittent bandwidth, and device mobility [1], [2]. Furthermore, differences in mobile computing platforms (heterogeneity) makes writing applications that target multiple devices a non-trivial process [3]–[5]. These challenges have slowed the adoption of mobile services.

The goals of our research are twofold: i) to create a middleware and framework to enable mobile services, and ii) to create a toolkit enabling developers to create mobile services without concern for platform and network heterogeneity. The first part of this research has been completed, resulting in the creation of Odin, our mobile service provisioning middleware [2], [5]–[10].

Odin provides a framework that developers can extend to create services for Android or iOS devices. However, using Odin by itself, developers would still be required to create separate versions of their services for each platform. Similarly, Odin itself would need to be re-implemented if support for an entirely different platform (e.g. Windows Phone 7) was required. This paper focuses on the design and development of OdinTools, an environment allowing cross-platform development of Odin-based services. OdinTools has been created using a Model-Driven Engineering (MDE) approach.

The remainder of this paper is organized as follows. Examples of related work in mobile service provisioning, service development, and cross-platform development are given in Section II. A brief overview of Odin is given in Section III. We describe and justify our OdinTools metamodel in Section IV, followed by its implementation in Section V. Finally, we conclude in Section VII following a discussion of future work in Section VI.

II. RELATED WORK

There has been significant work both in the areas of mobile Service Oriented Architecture (SOA) and cross-platform development. Furthermore, techniques such as Model-Driven Engineering (MDE) have been used with SOA to provide solutions to similar problems. The most relevant of these approaches are examined here.

A. Mobile Service Provisioning Approaches

Aside from Odin, other approaches to mobile service provisioning have been proposed. Relevant examples include
attempts to incorporate mobile computers into traditional SOAs. Cheng et al. [11] recognize the increased importance of privacy protection due to the nature of data traditionally stored on users’ mobile devices, while Natcheto et al. [12], Pawar et al. [13], Srima et al. [14], and Nokia [15] deal with limited, intermittent network connectivity by using a Gateway in the fixed network to manage mobile-to-client connectivity. Finally, Chou and Li [16] show how full duplex web services communication can be supported on low-powered hardware and maintained on unreliable mobile networks.

For a more comprehensive review of current work in this area, we refer to [6].

B. Cross-Platform Mobile Development

Several promising approaches to cross-platform development have been proposed in the literature. XMLVM, developed by Puder and Yoon [17], allows Android developers to cross-compile their mobile applications to other platforms. Currently, iOS and Palm-Pre devices are supported. Android applications written in Java are compiled normally into bytecode. The bytecode is then transformed into an XML representation using a custom tool. XSL stylesheets then transform this XML representation into code for the target language. The result is code written for the target platform that emulates the Java stack machine. Instructions are emulated on a bytecode level.

Titanium Developer [18] and Open-Plug ELIPS [19] are commercially available cross-platform development tools that allow developers to write mobile applications using a scripting language (JavaScript and ActionScript, respectively). The scripting languages are augmented with mobile-specific APIs such as GUI development and location-awareness. Using Titanium, the developer’s code is combined with an application skeleton for the target platform, resulting in an application that runs the developer’s code using a platform-specific engine (WebKit for iOS devices, Rhino for Android systems). ELIPS takes this a step further, transforming the developer’s code into equivalent C code. Whenever the developer’s code invokes a mobile-specific API, these calls are transformed into calls to APIs developed and provided by Titanium and ELIPS vendors.

C. Use of MDE in Services Computing

Model-Driven Engineering (MDE) techniques such as Object Management Group’s Model-Driven Architecture (MDA) [20] promise to enable developers to abstract away differences in underlying platforms, focusing solely on the software’s functional requirements [21]. In services computing, MDE has primarily been used to provide interoperability between different service platforms, abstract away cross-cutting concerns such as security, and allow developers to model services and service interactions without concern for precise implementation details.

Mayer et al. [22], Srinivasanmurthy et al. [23], and Ullberg et al. [24] show how MDE can be used as an approach to Service Orchestration. When using formal models to specify service interactions, these can be validated using model checking to ensure correctness. Furthermore, visual notations can be more easily developed, enabling a faster development process. Furthermore, model transformations can be used to automatically generate the interaction logic from the model, in well-known languages such as BPEL, Java, or Jolie.

Dhanesha et al. [25] focus on single service creation, showing how models can be used to abstract away differences in underlying service platforms. Menzel and Meinel [26] show how models can be used to specify enforceable security policies for services, and how model transformations can be used to integrate this cross-cutting concern with the rest of the service architecture.

D. Discussion

The existing work in mobile SOA is promising. Odin adopts many of the ideas discussed in the literature and combines them into a comprehensive mobile service provisioning solution while providing support for the latest mobile platforms and providing additional features such as component relocation and extensible context-awareness (see Section III).

Recent work has enabled cross-platform native application development for mobile devices, but much of this work focuses on GUI-centric and service-client applications, rather than the mobile services we wish to create. Work with MDE in services computing has abstracted away differences in service platforms, but not differences in underlying hardware and operating systems, as is the case with today’s wide variety of mobile devices. A comprehensive cross-platform mobile service development toolkit would thus combine principles in both fields to offer a novel solution. This is the case with OdinTools, which uses an MDE approach to specify services as well as the Odin middleware itself. The model-based specification of both Odin and the developer’s service is then transformed into a platform-specific service implementation (see Section IV).

III. The Odin Middleware

Odin is a middleware designed to enable developers to create mobile services. Developers use and extend the provided framework to implement their services, while many challenges inherent in mobile service provisioning are addressed by Odin itself.

Odin is built upon the Jini service platform [27] and Jini Surrogate Architecture [28]. Jini is a Java-based SOA. Services can be discovered and consumed based on arbitrary metadata, while the Jini middleware provides fault tolerance in the form of multiple service registries (JLUS) and management of disconnected services. The Jini Surrogate Archi-
tecture (JSA) extends Jini with the concept of a Surrogate Host (SH). A SH can be used to allow resource-constrained devices to communicate in a Jini federation when they would normally lack the resources to do so. It serves as a container for one or more Surrogates. A Surrogate is a representation of a service running on a resource-constrained device. The Surrogate exposes itself as a Jini service, and communicates with the device via a custom Interconnect protocol. Figure 1 shows an overview of Odin, including the interactions between the Client, Registry, SH, Interconnect, and Device components.

A. Features

Odin extends the JSA in several key areas to provide a comprehensive mobile service provisioning solution, addressing mobile service reachability, availability, and scalability issues. A brief summary of its key features is provided here; for further information we refer to [2], [5]–[10].

1) Vertical Handover between Multiple Interconnect Channels: Odin supports communication over any available network interface, both IP-based (e.g., 3G, Wi-Fi) and non-IP-based (e.g., Bluetooth) [6]. Furthermore, Odin supports two paradigms for communication over these networks. In the case where the SH can directly contact the device, a full-duplex stream-based protocol is used. When the SH cannot contact the device directly (for example, when the device is behind a mobile network operator’s firewall), a COMET-based approach is used in which a long-held HTTP request by the device allows the SH to push client requests and data on the response [6].

Furthermore, Odin supports Vertical Handover between network interfaces. This occurs both proactively when a more suitable network is detected, and reactively in case of network failure [6]. Proactive handover increases scalability, while reactive handover increases availability.

2) Surrogate Migration: Using Odin, a device’s Surrogate can be migrated between any available SH in the cloud [7]. This is useful as hosts experiencing failure or heavy load can be avoided, increasing availability. Furthermore, network usage can be decreased by migrating to a SH closer to the Device.

3) Relocatable Components: Service logic written by developers can be executed either on the Device or Surrogate. Furthermore, data can be stored at either location [10]. Developers can manually specify where logic and data are to be located, or they can let Odin choose automatically.

4) Context Awareness: During operation, Odin collects Context information including battery life, available networks and bandwidth, Device and SH computational resources, and Device location [9]. This information is transformed into an OWL model [29] and reasoned upon to automate vertical handover, surrogate migration, and component relocation. Policies determining how Odin should react to changes are specified in SQWRL [30] and are configurable by developers. Furthermore, Odin’s context model is extensible, allowing developers to specify and respond to additional context information [8].

B. Evaluation

Odin’s performance in several key areas has been extensively tested. Performance evaluations of vertical handover, migration, and context reasoning can be found in [6], [7], and [9] respectively, and the benefits of component relocation can be seen in [10]. Furthermore, several Odin-based services have been developed, including parcel-tracking software [8] and a social networking application [31], and work is underway with the University of Auckland Clinical Trials Research Unit to adopt Odin for the purposes of m-health and patient monitoring. Battery life, network utilization, and computational power have been shown to improve compared with a mobile device hosting a simple web server. This is due to intelligent use of available networks, and augmentation with external computational power (the SH).

IV. ODIN TOOLS METAMODELS

OdinTools uses an MDE approach, both i) to allow cross-platform service development, and ii) to allow Odin to be supported on new platforms in the future. To support both these goals, we have adopted a multi-level model hierarchy, as shown in Figure 2. Service developers define their mobile service with a Service Model (SM). Odin middleware features are introduced in the Odin Framework Model (OM). Abstract platform features used to implement Odin services are introduced in the Platform-Independent Model (PIM). Details of how specific platforms realize the abstract PIM are given in the Platform-Specific Model (PSM), provided by platform developers. Finally, the implementation of the
service for specific platforms can be generated. Each of these stages in the hierarchy is explained in more detail in this section.

![Diagram of OdinTools Model Hierarchy]

**Figure 2. Overview of OdinTools Model Hierarchy**

### A. Service Metamodel

Figure 3 presents an overview of the metamodel that service developers use to model their services. All objects in Odin-based services are known as *Entities*. An *Entity* can contain arbitrary data, known as *properties*. Odin’s Context-Awareness framework uses *ContextItems*, while *Services* can be both provided and consumed. Service has a *ServiceDescription* which provides enough information to uniquely identify it, and a proxy that can be used to invoke the service operations. Services support both request-reply semantics (via *invokeOperation*) and publish-subscribe semantics (via *subscribe*). Developers may also model a *User Interface (UI)* that will appear on the mobile device. They do this by defining instances of *UIComponents*. Developers model their own Entities, ContextItems, Services, and UIComponents by extending the corresponding metamodel classes. Some common Entities, ContextItems, and UIComponents are already provided by OdinTools.

The metamodel provides three utility classes, able to be invoked by all Odin-based services. A *DataStore* provides a simple abstraction of persistent storage, allowing developers to associate arbitrary data with a key, and recall it later using that key. A *ContextSource* utility allows developers to subscribe to changes in any context information, as well as publish their own context items. Any Odin-based service can listen to changes in context of any other Odin-based service. Finally, a *Registry* allows the developer’s service to be exposed to the outside world, while also allowing developers to consume other Services.

### B. Odin Metamodel

The SM allows the developer to specify the logic of a mobile service without concern for the underlying middleware, OS, or hardware. We need to generate a model at a lower level of abstraction to show how the developer’s service is realized by the Odin middleware. We achieve this by transforming the developer’s SM into a model conforming to *Odin’s Metamodel*. This new model, known as the OM, is generated automatically by OdinTools. Its metamodel is shown in Figure 4.

The Odin metamodel describes Odin’s features, as described in Section III. Service logic is decomposed into *DeviceService* and *Surrogate* components. Currently this decomposition is achieved through the use of *tags* on each of the developer’s service methods specifying where the method is to be placed. As described in Section III-A3, the runtime location of the components can change. Communication between the DeviceService and Surrogate components are achieved via *Messages* containing *Data* passed through an *Interconnect*. The *SurrogateHostDiscovery* utility allows the DeviceService to discover a *SurrogateHost*, which executes the Surrogate and exposes it as a Service. A *ContextProcessor* utility, located on the SurrogateHost, controls the Interconnect and Surrogate, commanding them to perform migration and handover when necessary.

When transforming the SM to the OM, the following mappings are defined:

1) ContextItems and ContextSource invocations are transformed to interactions with Odin’s ContextProcessor.
2) Each method in the Service is transformed to logic inside *handleSurrogateRequest*, *handleAsyncMessage*, or *handleDeviceRequest* in the Device or Surrogate, depending on a tag provided by the developer.
3) Invocations to the Registry are transformed to interactions with the SurrogateHost.
4) UIComponents defined by the SM are not transformed to OM elements - they are transformed directly to PIM elements (Section IV-C) as Odin has no knowledge of any user interfaces.

### C. Platform-Independent Metamodel

The OM allows us to see how the service modeled by the developer is realized by the Odin middleware. At the next lowest level of abstraction, the Platform-Independent Model (PIM), we see how Odin itself is realized by an abstract platform. Figure 5 shows the main metamodel elements introduced at this stage. The Device-side elements are shown, as the Surrogate-side model shows little change from the OM. We have modeled the Device-side elements after an Android application as the authors are most familiar with this paradigm.

Each Odin service is packaged into an *Application* consisting of one or more *Activities* and one *Service*. Each Activity represents one screen on the user interface and contains one or more *Views* (Android’s UI components) displaying information to the user. The Service contains all the OM’s DeviceService logic and uses a *NetworkInterface* utility to create *NetworkConnections*. Messages from the OM are
transformed into byte arrays and written to / read from these connections, thus emulating the OM’s Interconnect element.

D. Platform-Specific Metamodel

Once the PIM has been generated, a Platform-Specific Model (PSM) can be generated from which the implementation artifacts can be created. The metamodel to use for this transformation depends on which platform the developer is targeting. A single Platform-Specific Metamodel corresponds to a single platform.

Currently, a single PSM has been created, specifying an Android Device plus a Java-based Surrogate Host. Its components are similar to those shown in Figure 5 (in fact, the PIM was originally modeled from Android).

V. IMPLEMENTATION

Currently, an initial prototype of OdinTools has been implemented, allowing developers to create Service Models. The prototype has been implemented using Marama, an Eclipse plug-in. Marama was designed at the University of Auckland and is intended to provide a rapid prototyping environment for multi-view, multi-user diagramming applications and domain-specific visual languages [32]. Its ease-of-use for prototyping purposes was the primary motivator behind choosing this tool.

Figure 6 shows a screenshot of the OdinTools prototype. Developers drag and drop shapes from the lefthand panel onto a drawing surface. These shapes represent a Service’s components (ContextItems, Entities, ServiceDescriptions,
Services, etc). Developers may also drag EntityInstanceShapes onto the surface to define instances of various Entity classes (see “me” and “friends” in Figure 6). When the developer clicks on the Service entity, a code window appears allowing the developer to write the business logic for the service. Currently, the developer writes in Java, and this is transformed into another language if necessary (e.g. Objective-C for iPhone service development).

Figure 6 also shows the operation of the ContextManager utility. In the screenshot, the developer has defined a context rule that binds the value of me.location to the context item Context.DeviceLocation. DeviceLocation isn’t visible on the drawing surface itself as it is built into Odin and thus already provided by OdinTools.

A. Use Case – A Social Networking Service

As an example of how a service is developed using OdinTools, consider a scenario in which a service developer wishes to create a social networking service. This service has the following functional requirements:

- **P2P-style service:** instances of the service running on different devices may contact each other.
- **A map** is visible on a user’s mobile device (similar to Google maps).
- The map shows the **location** of the user’s friends (i.e. peers).
- The map will be **updated in real-time** as the user’s friends move around.

Using OdinTools, these are the steps the developer follows to create a mobile service based on the metamodel introduced in Section IV:

1) Upon creating a new service project, the developer chooses a name for their service (in this case, **SocialNetworkService**). This is defined by the “SocialNetworkService” shape on the drawing surface (see Figure 6).

2) The developer next defines the Entities describing the types of data in the service. In this case, the types to be created are **Person** and **Location**. A Person has a **name** (type: string) and a **location**. A Location has a **latitude** and **longitude** (type: double). String and double are provided by OdinTools, while Person and Location are defined by the developer by dragging and dropping EntityShapes onto the surface and filling out their properties (these shapes are shown in Figure 6).

3) The social networking service needs to store one Person representing a user, and one list of People representing the user’s friends. The user’s representation is called “me”, while the friends list is called “friends”. The developer defines these concrete variables by using EntityInstanceShapes (“me” and “friends” are visible in Figure 6).
4) The service needs to expose the user’s details (represented by “me”) to clients. The developer accomplishes this by defining a new service method, called getLocation() as part of the model’s Service object. Once done, the method body can be filled in on the code window while the Service shape is selected.

5) A user must be able to identify an instance of the social network service as belonging to a friend, as opposed to a stranger. A Service has metadata which can be used for this purpose. In this case, the developer uses the person’s name as the metadata. The developer defines this in a Service’s ServiceDescription (“SocialNetworkService metadata” in Figure 6).

6) When the user’s device location changes, the value of “me.location” needs to be updated. Odin supports obtaining GPS data as part of its context framework. The developer can set up a relationship between “me.location” and the provided context information. The developer uses a rule shape (“ContextManager:” in Figure 6) to do this.

7) A UI needs be defined to display the locations of the user’s friends. Currently, the UI is scripted in XML, with no graphical editor.

8) When a friend’s location changes, the UI needs to be updated. In the UI XML file, the developer sets up a binding tag, to bind a specific element of the UI to certain values. In this case, the developer binds the Markers property of a GoogleMapsComponent to [SELECT location FROM friends] – OdinTools supports SQL-like binding syntax similar to how Language Integrated Query (LINQ) functions in .Net languages.

9) The social networking service needs to be able to discover other instances of the service running on different devices, belonging to a user’s friends, in order to receive updates when those friends’ locations change. To do this, the developer specifies instances of SocialNetworkService (“friendsServices” in Figure 6). The developer then populates this list of services by dragging a rule between the friends list and the friendsServices list, telling the Registry to search for services with metadata matching the friends’ names.

VI. Future Work
Currently, the OdinTools’ Service Metamodel, Odin Metamodel, and Platform-Independent Metamodel have been specified, along with a Platform-Specific Metamodel for Android devices. Furthermore, a prototype visual metamodel has been implemented using Eclipse and Marama, allowing developers to specify their Service Model. Currently we are working on the transformations that will generate an Odin-based service from the developer’s model. In addition, we are investigating the creation of a second visual language to aid platform developers in enabling support for new mobile platforms. This will enable developers to specify Platform-Specific Models in the same way they currently specify Service Models.

Following the development of the prototype that fully supports Odin-based service development, we will conduct a usability analysis and use the results to refine the development. This may include switching to a different, more robust or expressive toolset, such as GMF or IBM's Rational software.

VII. Conclusions
To address mobility issues associated with the provisioning of mobile services, we have developed Odin, a middleware which masks complexity, allowing rapid development of mobile services. Odin, however, does not allow cross-platform development, which is an important concern with today’s wide variety of mobile devices. To solve this problem, we have designed OdinTools - a model-driven toolkit for cross-platform development of mobile services. Leveraging appropriate metamodels, a prototype has been implemented in Eclipse and Marama that allows developers to model mobile services in a platform-independent manner. We are currently working on transformations between levels of the model hierarchy which will allow full Odin-based service implementations to be specified automatically.

REFERENCES


