Dynamic and Automated Product Derivation for Consumer Electronics Software Applications

Ricardo E. V. de S. Rosa, Vicente F. de Lucena Jr., *Member*, IEEE, Lucas C. Cordeiro, and João E. Chaves Filho

**Abstract — ***Software Product Lines (SPL) is an efficient software engineering approach for dealing with reusable components in products that not only share common features, but also support specific functionalities that satisfy a particular market segment. This approach is interesting for the consumer electronics industry, particularly for mobile device applications. Despite having a significant common core, software applications developed for that domain have to be frequently adapted to different device features, such as operating systems and screen resolution. Thus, developers need to select proper software components to suitably compose the applications for each new device in a family of devices. In this paper, an approach that is able to customize consumer electronics software applications for different devices, in a dynamic and automated way, is presented. It results in a tool called AppSpotter that composes applications by selecting software components according to the features of each target device. To check the tool's performance, a set of experiments were realized in order to simulate different scenarios with up to 10,000 components* [1].

Index Terms — Product derivation, Dynamic software product lines, Mobile applications, Dependency injection.

I. Introduction

Since diverse consumer electronic devices started providing interactive services, the amount of applications and possible uses for these devices grow exponentially [1]. This trend has been established throughout recent years supported by the increasing evolution of existing infrastructure and the even more powerful new generation of devices and applications [2], [3]. In fact, the development of software for such system is an even more challenging task [4], [5].

The growing need for providing support for a variety of platforms available in the market (from low-end to high-end devices) is perhaps one of the most important challenges faced by consumer electronics software application developers [6], [7]. These developers need to produce several versions of the same application in order to deal with particular features. Indeed, the variation in features among devices suggests that portability requirements play an important role, as it is not economically viable to produce software only for a few devices (i.e., a small fraction of customers) [8]. Moreover, the poor adaptation of content for consumer electronics device might negatively affect users' Quality of Experience (QoE) [9]. For example, an application designed to smartphones may face problems related to screen size and resolution when running on a tablet. Therefore, one important requirement is to adapt the content according to the capabilities of target devices [10]. For such cases, Software Product Lines (SPL) seems to be a promising development technique, since it explores the similarities and diversities among related products.

In fact, SPL Engineering (SPLE) is a software engineering approach focused on improved productivity and efficiency, i.e., it reduces costs and time-to-market while improving the quality and reliability of the resulting products [11], [12]. SPLE aims at building a unique platform of software-related assets to be used during the development of individual products. As a result, a family of related products can be built by large-scale reuse of that unique platform.

Several authors have addressed SPLE, in its various aspects, to the consumer electronic devices domain and most specifically to the mobile devices domain [8], [13]-[15]. One of the most frequently discussed topics is the large number of features present in those devices. The variation of these features may result in a significant amount of variability in software. That is, besides market or users needs, the selection of alternative versions of software components also depends on technical limitations that include factors such as device capabilities, e.g., sensors, processing power, screen resolution, communication technologies, and operating systems [16]. Hence, the development of mobile applications becomes even more complex and the process of selecting proper software components becomes error-prone, time-consuming, and manually impracticable [17]. In addition, it is difficult to anticipate all possible versions of applications and it is improbable that applications specially designed for one device can be fully compatible with a different one without adjustments. These problems bring about the need for an approach to address the construction of software applications for this domain in a dynamic and automated way.

Product Derivation (PD) in SPLE refers to the construction of individual products from the software-related assets of a SPL [17]. It includes the selection, composition, and customization of these assets to deal with a specific SPL product and to satisfy the customer's requirements [18]. The

[1] This work was developed at Electronics and Information Technology R&D Center (CETELI) at the Federal University of Amazonas (UFAM). The authors are with UFAM and would like to thank the following institutions: FAPEAM, CAPES and CNPq for their financial support.

Ricardo E. V. de S. Rosa is with the Graduate Program in Electrical Engineering (PPGEE) at UFAM, Manaus, Amazonas, Brazil (e-mail: ricardoerikson@ufam.edu.br).

Vicente F. de Lucena Jr. is with the PPGEE, PPGI, and CETELI at UFAM, Manaus, Amazonas, Brazil (e-mail: vicente@ufam.edu.br).

Lucas C. Cordeiro is with the PPGEE and CETELI at UFAM, Manaus, Amazonas, Brazil (e-mail: lucascordeiro@ufam.edu.br).

João E. Chaves Filho is with the PPGEE and CETELI at UFAM, Manaus, Amazonas, Brazil (e-mail: jo_edgar@ufam.edu.br).

dynamic and automated PD associated with Dynamic SPL (DSPL) techniques [19] seems to be a promising approach for the handheld domain. It can be used for building mobile applications through a combination of existing software artifacts by adapting them to the features that are present on each mobile device without prior knowledge of the platform, or adapting the software according to the resources that are available on the consumer device using techniques based on dynamic resources management [20].

In this paper, an approach to support the automated composition of software applications will be presented. This approach results in a tool called AppSpotter that uses DSPL concepts for the dynamic and automated derivation of mobile applications. To achieve this, AppSpotter performs five steps in an automated way: (1) identifies the target devices within the range of the local wireless communication technology; (2) captures the features from these devices; (3) decides if it is possible to build applications that meet the captured features; (4) builds adapted versions of the desired applications considering the features from each device; and (5) sends the customized versions of the applications to each target device.

This paper is organized as follows: a background of the concepts related to the proposal is provided in Section II, an overview of the proposed approach is described in Section III, Section IV contains the AppSpotter's architecture, and the implementation of the AppSpotter tool is presented in Section V, experimental results are discussed in Section VI, and concluding remarks are presented in Section VII.

II. BACKGROUND

In this section, the main concepts of Product Derivation (PD) in SPL that underpin this proposal are described. Section II.A presents an overview of the SPL and its main concepts. Section II.B describes the concepts related to Dependency Injection (DI) and data-driven composition (DDC). In addition, an overview on Linear-time Temporal Logic (LTL) is given in Section II.C.

A. *Software Product Lines*

The transition from single systems development to product families development seems to be a clear trend in software engineering [11], [12]. Product families were defined as sets of programs that have so many features in common that it is worthwhile to analyze the commonalities before exploiting the existing variability (i.e., the features that differentiate them). The main purpose of analyzing the common features is to reuse software artifacts over the entire family of programs, thus improving software quality while significantly reducing time-to-market and maintenance costs [21].

In SPLE, product families are developed in a two stage development process [12], [22]. In the first stage, which is called domain engineering, there is an up-front investment to analyze the family and to build up the artifacts that comprise the platform, i.e., the foundation for creating new products. In the second stage, which is called application engineering, instances of the software products are constructed by exploiting the product line variability and reusing the artifacts that were previously obtained during the domain engineering phase. These instances are combined with application-specific artifacts in order to derive individual products for a particular market segment or customer needs [17], [23].

One possible way to derive individual software products in the mobile devices domain is by using annotative approaches to configure different features [24]. A typical example of annotative approaches is the #ifdef and #endif statements found in programming languages like C and C++.

B. *Dependency Injection and Data-Driven Composition*

SPL development requires the deployment of well designed components, which cover the scope of the product line. In addition, these components must also provide the possibility for developing functionalities beyond the normal SPL scope. In this context, high cohesion and loose coupling are software attributes that play an important role in reducing the dependency among components and facilitating the integration of new ones. These two attributes can be achieved by following the Dependency Inversion Principle (DIP).

Using this approach, concrete implementations can manageably be replaced by alternative implementations without affecting the high level modules. DI, which is a form of DIP, is a compositional approach that provides a flexible way to indirectly wire the software components together [25]. It is a common feature of frameworks where the application objects are dynamically instantiated and configured for use [26]. A framework is responsible for resolving the dependencies and injecting them into the application without explicitly hard coding in the application classes.

Data Driven Composition (DDC) is a compositional approach for specifying dependencies in component-based systems in which a designated object holds additional knowledge about the correctness of the composition. By using DDC, the relationship among objects or attributes contained within each object is specified through meta-data either in XML or in any appropriate scripting language.

C. *Linear-Time Temporal Logic*

Linear-time Temporal Logic (LTL) is a commonly used specification logic for expressing temporal properties of systems [27]. In this paper, LTL is used to specify (see Section IV.D) and verify (see Section VI.B) some temporal properties of the PD system, e.g., responsiveness or mutual exclusion.

LTL extends propositional logic by including temporal operators that allow modeling time as a sequence of states, extending infinitely, according to the following definition:

Definition 1. *The syntax of LTL is defined over a set of atomic propositions, logical operators and temporal operators in the following form:*

$$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2)$$
$$\mid (\phi_1 \rightarrow \phi_2) \mid (X\phi) \mid (F\phi) \mid (G\phi) \mid (\phi_1 U \phi_2) \mid (\phi_1 R \phi_2)$$

where the symbols $\top$ and $\bot$ represent true and false, respectively. p is any atomic proposition. The logical operators are: negation (¬), conjunction (∨), *disjunction* (∧), *implication* (→) and *biconditional* (↔).

The connectives *X*, *F*, *G*, *U* and *R* are called temporal operators. *X* means "ne*X*t state", *F* means "some *F*uture state", and *G* means "all future states (*G*lobally)". The next two, *U* and *R*, are called "*U*ntil" and "*R*elease", respectively.

Software systems that are specified using LTL can be modeled as state transition systems by means of states and transitions. More formally:

Definition 2. *A state transition system $\mathcal{M} = (S, R, S_0)$ is defined by a set of states S, a set of transitions $R \subseteq S \times S$ and a set of initial states $S_0 \subseteq S$.*

A state transition system has a collection of states S, a relation R specifying how the system can move from state to state such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$. The semantics of an LTL formula is then defined along a computation path or simply a path $\pi = s_1 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$, which is an infinite sequence of states along $\mathcal{M}$ and represents a possible future of the system.

In LTL, a labeling function $L: S \rightarrow \mathcal{P}(Atoms)$ is assumed, mapping L from each state to the set of propositional variables represented by $Atoms$. $\mathcal{P}(Atoms)$ denote the power set of $Atoms$, a fixed set of atomic formulas (such as *p*, *q*, *r*,… or $p_1, p_2, p_3, \ldots$). Associated with each state *s*, one has a set of atomic propositions *L*(*s*) that are true for that particular state. *L* is just an assignment of truth values to all propositional variables. To formally describe the semantics of LTL formulas, Definition (2) is extended to include a labeling function $L: S \rightarrow \mathcal{P}(Atoms)$, so that $\mathcal{M}$ now becomes a tuple $\mathcal{K} = (S, R, S_0, L)$, which is called a Kripke structure.

Definition 3. *A Kripke structure is a tuple $\mathcal{M} = (S, R, S_0, L)$ where S is a set of states of $\mathcal{K}$, R is a set of transitions, S_0 is a set of initial states and $L: S \rightarrow \mathcal{P}(Atoms)$ is a labeling function, which defines for each state $s \in S$ the set L(s) of all propositional variables that belong to s.*

The set of all possible behaviors of a Kripke structure can be defined through the notion of computation trees:

Definition 4. *Let $\mathcal{K} = (S, R, S_0, L)$ be a Kripke structure over a set of variables $Atoms$ and $s \in S_0$ be a state. The computation tree for $\mathcal{K}$ is the following tree:*

1. *The root of the tree is labeled by the state* $\mathrm{s} \in \mathrm{S}_0$;
2. *The nodes of the tree are labeled by states in S*;
3. *For every node s' in the tree, its children are exactly $s'' \in S$, such that, $(s' \rightarrow s'') \in R$.*

Given Definition (4), a computation tree for a Kripke structure $\mathcal{K}$ can be identified as the set of its paths. A computation path π for $\mathcal{K}$ is a sequence of states $\mathrm{s}_1 \rightarrow \mathrm{s}_2 \rightarrow \cdots$ such that $(s_i \rightarrow s_{i+1}) \in R$ for all *i*. Additionally, if a sequence is finite, i.e., it has the form $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$, then there exists no state *s* such that $(s_n \rightarrow s) \in R$.

An intuitive interpretation of temporal operators over computation path formulas is as follows:

- $X\phi$ holds if ϕ holds at the next state on the path;
- $F\phi$ holds if ϕ eventually occurs at some future state on the path;
- $G\phi$ holds if ϕ holds globally at every state along the path;
- $\psi U\phi$ holds if ψ holds continuously until ϕ occurs;
- $\psi R\phi$ holds if either ϕ holds globally on the path or ψ occurs before the first state at which ϕ is violated.

LTL formulas are means for expressing properties of paths in computation trees, and are also conveniently used to discuss possible temporal behaviors of a system. For the understanding of the concepts described in this paper, the following properties are particularly important: *reachability*, *mutual exclusion*, and *responsiveness*. Reachability claims that a state is called reachable if there is a computation path from an initial state leading to this state. The mutual exclusion property must ensure that no two or more processes are allowed to be in the same critical section simultaneously. The responsiveness property is interested in verifying whether every request is eventually acknowledged in the system.

III. Overview of the AppSpotter Tool

The approach presented in this paper applies to the consumer electronics domain, most specifically to the mobile applications domain, as it uses the knowledge of device features to support the derivation of software applications. It is based on the concept of DSPL and seeks to manage the software variants at deployment-time through the dynamic and automated binding of software components.

Figure 1 presents an overview of the AppSpotter tool in which the product derivation process interacts with four illustrative mobile devices. In summary, the interaction takes place in five steps:

1. In the first step, the AppSpotter tool searches for devices that are within the range of the wireless communication technology and captures the features of each device found (the four devices in the figure). The captured features ($C1$, $C2$, $C3$, and $C4$) are then used in the next step.
2. For each device, the features are used to prune away the software components that are not compatible with it. If the remaining components are enough to build an application, then the process proceeds to the next step. Otherwise, the user is informed that it is not possible to produce an application for his/her device. In such a situation, developers can log information expressing their demands for new components to future use.
3. In the third step, the components that were selected for each device are configured by following the data-driven composition configurations (described further in Section V.B). These configurations are used to drive the application composition when wiring the components together. The result of this process is a configuration file for each target device.
4. In the fourth step, the source code of the software components is compiled to each device. The instances of the components are dynamically loaded according to the resulting configuration files from the previous step. As a result, one application for each target device is generated ($App1$, $App2$, $App3$ and $App4$).
5. The applications $App1$, $App2$, $App3$ and $App4$ are finally delivered to the devices that provided the features $C1$, $C2$, $C3$, and $C4$, respectively.

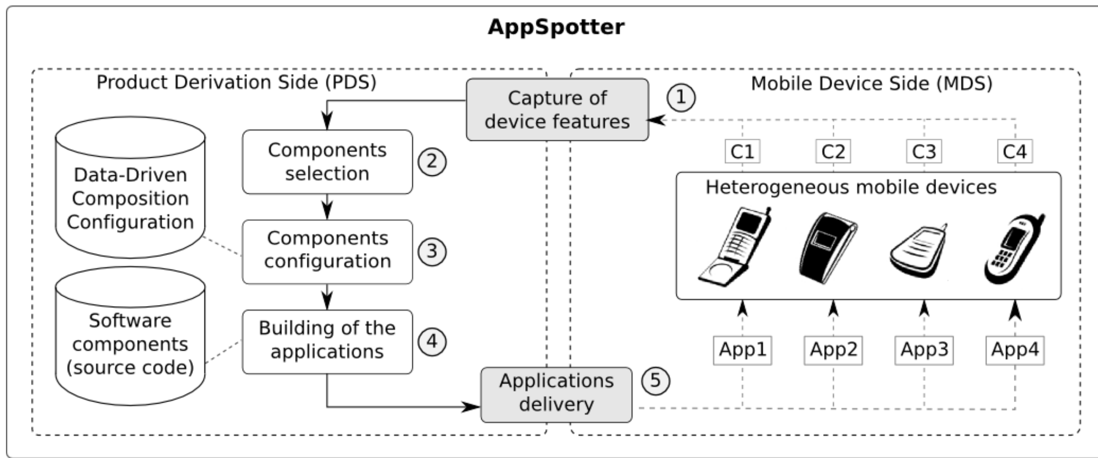

Fig. 1. Overview of the AppSpotter tool and its interaction with the mobile devices.

The result is a dynamic and automated mechanism to provide custom applications to feature-rich devices. Using this approach, the role of the developer is to implement the software components and design the architecture of the mobile application by following the design principles of the "strategy" pattern [28]. The developer must also specify what components are related to each device feature and how the components are wired to each other by using the data-driven composition configurations. The AppSpotter tool is responsible for providing a process to integrate the components based on a set of features, thus producing a custom application for each target device.

IV. AppSpotter's Architecture

The AppSpotter's architecture has two main parts: Product Derivation Side (PDS), detailed in Figure 2(a), and Mobile Device Side (MDS), detailed in Figure 2(b).

A. *Product Derivation Side (PDS)*

The PDS is divided into four parts: communication, components' selection, components' configuration, and application building.

1) *Communication*

The Bluetooth technology was chosen to allow wireless communication between PDS and MDS since this technology is supported by a wide range of mobile devices. Thereby, the components of the communication part are: *Services* and *Bluetooth*. The *Bluetooth* component is an interface that is used by other components to establish communication with mobile devices. The *Services* component uses the *Bluetooth* component to send information to and request information from mobile devices through a set of services as follows:

- *notifyDevices*. This service searches for nearby devices and informs about the availability of a custom application.
- *sendAppInformation*. The user can request additional information about the application mentioned by *notifyDevices* service. *sendAppInformation* is used to send this information.
- *receiveDeviceFeatures*. This service is used to receive the device features.

2) *Selection of software components*

The method for selecting the components intended to be reused for composing the application is based on a search engine for software components. Using search engines has proven to be very useful for organizing and retrieving software artifacts [29], [30]. The requirements for using each SPL artifact are stored and used to find the components that match those requirements.

The components of the selection part are: *Indexer* and *Searcher*. The *Indexer* component stores and organizes the SPL artifacts into a repository aiming to enhance the search performance. The *Searcher* component has a set of operations that are used to query over the repository. It receives a set of device features that are matched with the component requirements. If any given component meets the devices features, it is selected.

A query expression, which is used to find compatible software components, is created upon the device features. This expression is built by combining implicit logical disjunction ("*OR*" operator) and terms' exclusion ("-" operator).

The configuration of components is based on the concept of data-driven composition. These configurations are defined by developers and represent the composition of software components for the whole SPL, i.e., how the components are wired together to deliver the required functionality.

3) *Configuration of software components*

The Components Configurator component parses the data-driven configuration files and creates new files containing specific configurations for the set of components selected for each application. It provides flexibility during the composition of the software application since different implementations of components can be indirectly used without the need for changing the source code.

4) *Building the mobile application*

The construction of mobile application occurs after the configuration of software components. For each application, the *Builder* component receives both source code and dependency injection configuration, and performs the compilation and packing process. The result of this process is

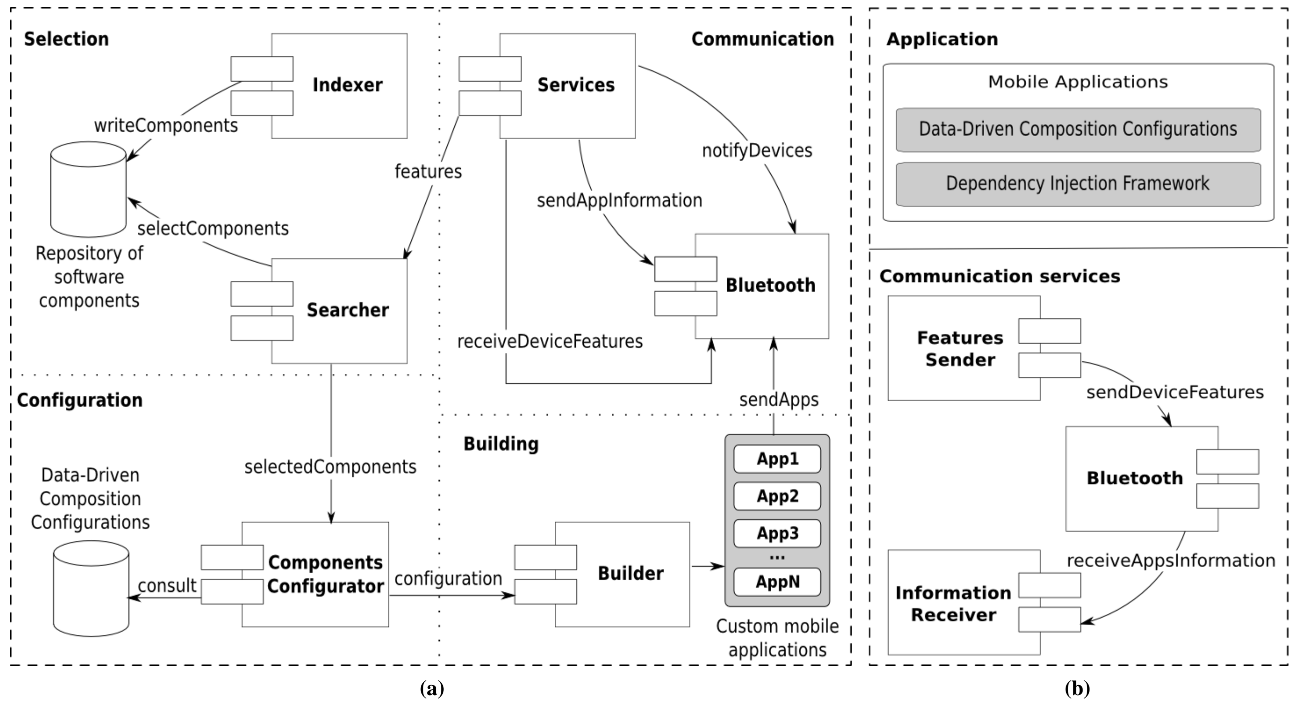

Fig. 2. Architecture presenting the components that are present in both sides of the AppSpotter tool: (a) Product Derivation Side – PDS, and (b) Mobile Device Side – MDS.

an application that can be installed and executed on the target mobile device. The resulting file is sent to the mobile device through the *Bluetooth* component, which is also responsible for sending the application to the target device through the service *sendApps*.

B. Mobile Device Side (MDS)

The AppSpotter's MDS is divided into two parts: communication services and application.

1) Communication services

A precondition for using the AppSpotter's approach is the presence of a communication service running on the device. When the device receives a connection request on this communication service, the user is notified about the existence of an available application. Thus, using the mobile device, a user can interact with the PDS to receive additional information about the application or request the delivery of a customized application.

As shown in Figure 2(b), the background service has three components: *Features Sender*, *Information Receiver*, and *Bluetooth*. The *Bluetooth* component has the same role as in PDS, which is to serve as interface for external communication. The *Features Sender* component uses the *sendDeviceInformation* method to send the device features to the PDS. The *Information Receiver* component requests additional information about the application by using the *receiveAppsInformation* method.

2) Application

On the mobile device, the components are dynamically loaded through a lightweight Dependency Injection (DI) framework embedded into each application. The created applications run on top of the platform layer using resources provided by the mobile device such as operating system and sensors. The components that compose the application are at the application layer, including the DI framework used to load these components at run-time. In fact, the DI framework can also be used to load both native and third party APIs to get access to the system resources.

C. Interaction Between PDS and MDS

PDS and MDS interact by means of Bluetooth communication technology, which is represented by the *Bluetooth* component shared by both parts. The interaction takes place in four distinct moments.

1) Notifying devices

Precondition. 1) At least one nearby device (within the range of the communication technology); and 2) The PDS needs to be ready for searching new devices.

The *Services* component has a *notifyDevices* service, which is responsible for searching nearby devices and notifying them about the availability of a custom application. Once this service finds a device, it requests a connection to the background service running on the device side. The background service then starts an application that shows two options: “receive additional information” and “download a custom application”.

2) Receiving information about the application

Precondition. 1) A mobile device was found and notified; and 2) The user has selected the option “receive additional information”.

If the user has selected the option “receive additional information”, the device uses the *receiveAppsInformation* service to request additional information from the PDS. When the *Services* component receives the request, it uses

sendAppInformation service to send information describing the application back to the device. The information is received by the *Information Receiver* component and is presented on the device's screen to the user.

3) Sending device features

Precondition. 1) A mobile device was found and notified; and 2) The user has selected the option "download a custom application".

When the user selects the option "download a custom application", the *Features Sender* component uses the *sendDeviceFeatures* service to request a connection and send the device features to PDS. When the request is received on the PDS side, the *Services* component uses the *receiveDeviceFeatures* service to receive the device´s features. These features are intended to be used in the PD process.

4) Sending application

Precondition. 1) The device features were received; and 2) A custom application was produced.

After receiving the device features, PDS starts the process to produce the application. At the end of the process, when an installable package is obtained, the PDS uses the *sendApps* service to send the application to the target device. The application is sent over a conventional file transfer protocol using Bluetooth connection.

D. State Transition System of the Derivation Process

Since the derivation process can be represented by a state transition system (as described in Section II.C), LTL was used to specify the temporal properties and to show how the system changes from state to state. Let $\mathcal{K} = (S, R, S_0, L)$ be a Kripke structure that represents the system (recall Definition 3).

Let $Prop = \{a, b, c, d, e, f, g, h\}$ be a set of atomic formulas. Each formula is denoted by one proposition, as follows:

- a: the system is searching for new devices;
- b: a new device has been found;
- c: the features of the new device are obtained;
- d: it is possible to create a valid application configuration (i.e., for the new device);
- e: the system is configuring a new application;
- f: the system is compiling the application;
- g: the system is sending the installable package of the application;
- h: the application has been sent.

Through LTL, some temporal properties of the transition system of the derivation process can be formally express. The first one is the responsiveness property. The system starts the process by searching for new devices and it continues searching until a new device is found. This can be expressed by the following formula:

$$G(a \rightarrow (a\ U\ b))$$

The second property that can be specified is related to reachability. At a certain point in the process, the system has to decide whether it is possible to continue the derivation process or it needs to restart. If the device features have been obtained and it is possible to create an application, then the system starts configuring the application. If it is not possible to create an application, the process is restarted and the system begins the search process again. This behavior can easily be expressed by the following formula:

$$G(\big((c \wedge d) \rightarrow e\big) \vee ((c \wedge \neg d) \rightarrow a))$$

A third property of the system is related to mutual exclusion. Since the system generates configuration files that are used to produce the application, it is recommended that these files are not handled by more than one process at a time. Thus, let e denote that a process is generating configuration files and let f denote that another process is compiling the application, a mutual exclusion property states that e and f never hold simultaneously. This behavior can be described by the following LTL formula:

$$G\neg(e \wedge f)$$

Another behavior related to responsiveness property is when the system is sending an application to the target device after compilation. The system must continue sending the application until it is fully sent. This property can be formally expressed by the following formula:

$$G(g \rightarrow (gU(\neg g \wedge h)))$$

The properties described for the state transition system in this section can be formally verified by using a model checking tool.

V. Implementation of the AppSpotter Tool

The initial prototype of this work deals with the development of applications that can be executed on devices compatible with the J2ME platform, Mobile Information Device Profile (MIDP), and Connected Limited Device Configuration (CLDC) [31]. However, the proposed approach does not require uniquely J2ME and the use of this technology does not subtract the tool generality. Thereby, the development of applications can also be targeted toward different mobile platforms. For example, some programming languages and modern application frameworks have the ability to examine and modify the structure and behavior of objects at run-time. This ability is called reflection and it is used in the implementation of the AppSpotter tool. As a result, languages such as Python and Objective-C, which support object oriented programming and reflection are fully compatible with the proposed approach.

A. Software Infrastructure

One free API was used [32] to implement the storage and retrieval capabilities of software components. In fact, the software components are stored into a repository, which enables their latter retrieval based on the actual needs expressed in a query expression. When the device features are captured, a query expression (combining these features) is generated by the proposed tool. The query is built by first including all components that meet the device features and

removing the ones that are not necessary. The removal is performed by using the minus signal ("-") in the query criterion. For example, if a device contains the features *screen* equals *360x640* and *accelerometer* equals *no*, the query expression will include the following criteria:

screen:360x640 -accelerometer:yes

This expression retrieves only components that are compatible with devices that have *360x640* screen resolution. Additionally, the expression excludes all components that require accelerometer sensors so that the search removes the components that are not compatible with the criteria and retrieves only those that meet the set of features.

The DI configurations of the SPL are specified by means of metadata in the XML format, which is easy to maintain and suitable for integrating with third party tools. An XML parser was developed to extract the data from the XML files. This data is used to generate new configuration files that are used during the instantiation of the components of each application on the mobile device.

The compilation and packing process of the application is performed using automated scripts. The target of compilation is composed by devices that are compatible with the MIDP and the CLDC. Then, the compiled binaries are packed into a*.jar* file which is ready to be installed on the device.

B. Data-Driven Composition

The use of data-driven composition allows specifying the dependency among the software components in a declarative way, which avoids tight coupling among components and facilitates the parsing of data. An XML-based approach was used to represent the dependency among the components by means of metadata, defined through an XML Schema Definition (XSD).

In the specification of components in XML, the root element of the configuration is *application* and it must contain the components that depend on each other in the SPL. The second element in the hierarchy is *component*, which describes the software components and contains two attributes: *id*, which is a unique identifier for each component, and *class*, which represents the class that must be instantiated. The *component* element consists of three elements:

- *composite*, which describes how the component must be composed. It has one attribute called *strategy*, which specifies the interface of the component.
- *dependencies*, which specifies a list of artifacts (through the *artifact* element) that the component needs to operate properly, like APIs.
- *constraints*, which specifies the requirements for using a given component. The *constraints* element is composed of a list of *feature* elements, which contain two attributes: *name*, the name of the feature that a device must have; and *value*, the value of the feature.

The goal is to model the dependency among the components using XML metadata. After selecting the components according to the features present in each device, the metadata that is not compatible with the device is removed during configuration process. Only components that are compatible with the device features will be processed by the DI framework, thus reducing the complexity of the application when running on the mobile device.

```
<device>
  <feature name="screen" value="360x640"/>
  <feature name="input" value="touch"/>
  <feature name="keypad" value="qwerty"/>
  <feature name="accelerometer" value="yes"/>
</device>
```

Fig. 3. Example of the XML file that is embedded into the device containing the features.

C. Dependency Injection on the Mobile Application

DI provides a flexible way to indirectly assemble the software components together [25]. By using DI in AppSpotter, the developer implements the dependency by declaring an attribute and specifying the composition through the *composite* element in XML.

At the code level, the dependency is implemented by using the strategy design pattern [28]. This pattern defines a common interface for alternative implementations of a given feature and uses polymorphism of object-oriented programming to access the needed implementation.

The dependency is automatically resolved at run-time by using lazy instantiation and its instance is assigned to the dependent component. There are two main advantages associated with this practice: (1) the source code becomes smaller and cleaner since it is not necessary to hard code the objects and their dependencies; and (2) fewer implementation errors since it is the responsibility of the framework to solve the dependencies. In addition, this implementation model contributes toward optimizing startup time, which is a desirable feature of consumer electronics applications [3], [33], [34].

D. Software Infrastructure on the Mobile Device

In the mobile device side, a set of functionalities was developed in order to allow the exchange of data with the AppSpotter tool. One of these functionalities is able to gather information about the mobile device features and send this information to the tool. To achieve this, an XML file containing a description of the features was embedded into the device. Figure 3 presents an example of this file where each feature contains a *name* and a *value* associated with it.

Another implemented functionality is the lightweight DI framework, which is packed into each application and is used to solve and instantiate the components at run-time, assigning them to the dependent components. While the prototype implementation of this lightweight DI framework was implemented using Java language, alternative versions can be implemented by using other programming languages to support other platforms of interest; however, they need to support dynamic class loading in order to load the needed components at run-time. Modern object-oriented programming languages and frameworks (e.g., C++, Objective-C and Python) frequently support this feature by loading classes directly from binary or shared library files.

VI. Experimental Results

To motivate the need for using AppSpotter in consumer electronics applications, a scenario where users can buy movie tickets through their mobile devices right from home or anywhere else will be explained. Aiming to reduce queues for ticket-booking, a movie theatre would like to deploy such system online. Someone wanting to watch a movie can use his/her mobile device to check the time-table and tickets' availability. After choosing the desired movie, this person can purchase the ticket and receive a key to enter in a booking machine in order to receive the printed tickets.

The system consists of an application that must be installed on the client side (i.e., mobile devices). Through this application, the user is able to connect to an online server aimed at requesting information related to the movie of interest and ordering the tickets. The application must suit the device configuration. That is, for each device configuration, an alternative version of the application must be deployed aiming to fully explore the device capabilities (e.g., screen size). In this way, a user with a tablet would probably have an alternative version of the application, which is different from the one installed on a conventional smartphone.

A set of components was developed in order to implement the scenario described above. They were used as a prototype of the application on the mobile devices. Each component has alternative versions for different device configurations.

A. *Performance of the components selection*

A performance experiment focused on the selection time to find the components that met a given device configuration was executed. This experiment was performed on an idle PC with a dual-core processor, 2.4 GHz of clock speed and 4 GB RAM, running a UNIX-based operating system.

During the mapping of components and their dependencies, it was ensured that a given alternative implementation of a component could not be simultaneously selected by two or more different device configurations. By doing this, it was avoided conflicts during the instantiation of a component in the final application. A set of test cases ranging from 20 to 10,000 components was created. These test cases were used to create two scenarios: worst case selection time (WCST) and best-case selection time (BCST). WCST happens when all of the SPL components have at least one requirement that is part of the device configuration. BCST happens when, given a device configuration, only the needed components have the necessary requirements for selection. About 200 rounds of experiments were executed for each test case in each test scenario. The performance results can be seen in Figure 4, where it is presented the average time to perform a selection based on predefined criteria, i.e., device configuration. The results show a fast selection time even in the worst scenario.

B. *Verification of the LTL temporal properties*

Given the specifications of the LTL temporal properties of PD process in Section IV.D, some experiments to verify the consistency of these properties were realized. A symbolic model checker tool was used [27]. This is an open source tool

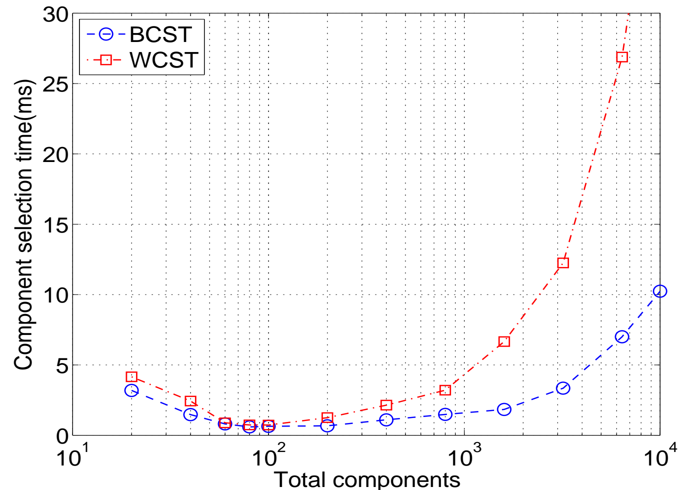

Fig. 4. Performance results for best and worst case selection time.

actively supported by the software engineering community that provides a language for implementing the transition relations of models and directly checks the validity of LTL formulas on these models. As a result of the experiment, it was possible to verify that the LTL properties hold for the state transition system of the product derivation process (Section IV.D). The used model checker tool took less than one second to verify the specified LTL formulas.

VII. Conclusions

The automated selection and composition of software components for the creation of consumer electronics software applications is very useful when the SPL contains a large number of components. This high number of components may result in a significant number of software variants, and solving this variability manually is a time-consuming and error-prone process. Taking this into account, a tool called AppSpotter was idealized and developed. This tool can be used for selecting software components in a dynamic and automated way by using the device features along with a search engine of components. The tool can also be used for composing software applications using only the components that are compatible with the electronic device features.

The main results that were achieved with this work are:

- A mechanism for selecting software components compatible with the particular features of a mobile device.
- A data-driven approach based on XML for defining the dependencies of software components of SPL and how they are wired together to compose the application.
- A lightweight DI framework to instantiate the components at run-time.
- The possibility of building mobile applications by using the device features rather than the device model.

A set of experiments was realized in order to check the performance of the tool. The experiments included the selection of components in scenarios with up to 10,000 artifacts. The tool achieved very low selection times, in the worst case scenario, the selection time takes less than 50ms, while in the best case scenario, it takes about 10ms.

REFERENCES

[1] M. Vidakovic, T. Maruna, N. Teslic, and V. Mihic. “A Java API interface for the integration of DTV services in embedded multimedia devices*,” IEEE Trans. Consumer Electron.,* vol. 58, no. 3, pp. 1063-1069, Aug. 2012.

[2] N. Kuzmanovic, V. Mihic, T. Maruna, M. Vidakovic and N. Teslic, “Hybrid broadcast broadband TV implementation in Java based applications on digital TV devices*,” IEEE Trans. Consumer Electron.,* vol. 58, no. 3, pp. 1056-1062, Aug. 2012.

[3] H. Kasai, “Embedded middleware and software development kit for area-based distributed mobile cache system*,” IEEE Trans. Consumer Electron.,* vol. 59, no. 1, pp. 281-289, Feb. 2013.

[4] V. F. de Lucena Jr., J. E. Chaves Filho, N. S. Viana, and O. B. Maia, "A home automation proposal built on the Ginga middleware and the OSGi framework," *IEEE Trans. Consumer Electron.*, vol. 55, no. 3, pp. 1254-1262, Aug. 2009.

[5] V. F. de Lucena Jr., N. S. Viana, O. B. Maia, J. E. Chaves Filho, and W. S. da Silva Jr, "Design an extension API for bridging Ginga iDTV applications and home services," *IEEE Trans. Consumer Electron.*, vol. 58, no. 3, pp. 1077-1085, Aug. 2012.

[6] S. Spinsante, and E. Gambi, “Remote health monitoring by OSGi technology and digital integration*,” IEEE Trans. Consumer Electron.,* vol. 58, no. 4, pp. 1434-1441, Nov. 1998.

[7] F. Almenárez, P. Arias, D. Díaz-Sánchez, A. Marín and R. Sánchez, “fedTV: Personal networks federation for IdM in mobile DTB*,” IEEE Trans. Consumer Electron.,* vol. 57, no. 2, pp. 499-506, May 2011.

[8] V. Alves, G. Santos, F. Calheiros,V. Nepomuceno, D. Pires, A. C. Neto, and P. Borba, “Beyond code: handling variability in art artifacts in mobile game product lines,” *Proc. of the Workshop on Managing Variability for Software Product Lines: Working With Variability Mechanisms (SPLC 2006)*, pp. 124-132, 2006.

[9] B.-Y. Lee, “Provisioning of adaptive rich media services in consideration of terminal capabilities in IPTV environments,” *IEEE Trans. Consumer Electron.,* vol. 57, no. 3, pp. 1120-1127, Aug. 2011.

[10] N. Kim, J.-Y. Yoo, N. L. Kim, J. W. Kim, “A visual-sharing switching device supporting programmable in-network content adaptation,” *IEEE Trans. Consumer Electron.,* vol. 58, no. 2, pp. 413-418, Jul. 2012.

[11] K. Pohl, G. Böckle, and F. J. Linden, *Software product line engineering: foundations, principles and techniques*, 1st ed., Springer: New York, 2005, pp 3-18.

[12] F. J. Linden, K. Schmid, and E. Rommes, *Software product lines in action: the best industrial practice in product line engineering*, 1st ed., Springer-Verlag: New York, US, 2007, pp. 3-20.

[13] V. Alves, “Identifying variations in mobile devices,” *Journal of Object Technology*, vol. 4, no. 3, pp. 47-52, Apr. 2005.

[14] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, and S. Hallsteinsen, “Modeling of component-based self-adapting context-aware applications for mobile devices,” *Software Engineering Techniques: Design for Quality* (IFIP), Springer: Boston, US, vol. 227, 2007, pp.85-96.

[15] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko, “Automatically composing reusable software components for mobile devices,” *J. Braz. Comp. Soc.*, vol. 14, no. 1, 2008, pp.25-44.

[16] R. Rosa and V. Lucena Jr., “Smart composition of reusable software components in mobile application product lines,” *Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering*, ACM: New York, USA,, pp. 45-49, 2011.

[17] S. Deelstra, M. Sinnema, M., and J. Bosch, “Product derivation in software product families: a case study,” *J. Syst. Softw.*, Elsevier, vol. 74, no. 2, pp. 173-194, Jan. 2005.

[18] R. Rabiser, P. O'Leary, and I. Richardson, “Key activities for product derivation in software product lines,” *J. Syst. Softw.*, Elsevier, vol. 84, no. 2, pp. 285-300, Feb. 2011.

[19] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, “Dynamic software product lines,” *Computer*, IEEE, vol. 41, no. 4, pp. 93-95, Abr. 2008.

[20] C. Hentschel, R. J. Bril, Y. Chen, R. Braspenning and T.-H. Lan “*Video Quality-of-Service for consumer terminals – a novel system for programmable components,*” *IEEE Trans. Consumer Electron*., vol. 49, no. 4, pp. 1367-1377, Nov., 2003.

[21] P. Clements and L. Northrop, *Software product lines: practices and patterns*, 3rd ed., Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002, pp. 29-50.

[22] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*, 1st ed., Addison-Wesley Publishing Co.: New York, NY, USA, 2000, pp. 17-81.

[23] R. Rabiser, P. Grünbacher, and D. Dhungana, “Requirements for product derivation support: results from a systematic literature review and an expert survey,” *Inf. Softw. Technol.*, Elsevier, vol. 52, no. 3, pp. 324-346, Mar. 2010.

[24] L. M. Nascimento, E. S. Almeida, and S. R. L. Meira, “A case study in software product lines - the case of the mobile game domain,” *Software Engineering and Advanced Applications*, (Euromicro Conference), IEEE Computer Society: Los Alamitos, CA, USA, pp. 43-50, 2008.

[25] M. Fowler, “Module assembly,” *IEEE Softw.*, IEEE, vol. 21, no. 2, pp. 65-67, Mar. 2004.

[26] M. Mattsson, J. Bosch, and M. E. Fayad, “Framework integration problems, causes, solutions,” *Commun. ACM*, ACM, vol. 42, no. 10, pp. 80-87. Oct. 1999.

[27] M. Huth and M. Ryan, *Logic in computer science: modelling and reasoning about systems*, 2nd ed., Cambridge University Press: New York, NY, USA. 2004, pp. 187-206.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, 1st ed., Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1995, pp. 315-324.

[29] H. I. Alsawalqah, K. S. Abotsi, and D. H. Lee, “An automated mechanism for organizing and retrieving core asset artifacts for product derivation in SPL,” *Proc. of the 2nd Int. Conf. on Interaction Sciences Information Technology, Culture and Human*, ACM Press: New York, NY, USA, pp. 80-485, 2009.

[30] O. Hummel, W. Janjic, C. Atkinson, “Code Conjurer: Pulling Reusable Software out of Thin Air,” *IEEE Software*, vol. 25, no.5, pp.45-52, Sep.-Oct. 2008.

[31] C. E. Ortiz and Giguere, E., *Mobile information device profile for Java 2 micro edition*, 1st ed., John Wiley & Sons: New York, NY, USA, 2011, pp. 12-26.

[32] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in action, second edition: covers Apache Lucene 3.0*, 2nd ed., Manning Publications Co.: Greenwich, CT, USA, 2010, pp. 74-109.

[33] H. Jo, H. Kim, J. Jeong, J. Lee, S. Maeng, “Optimizing the startup time of embedded systems: a case study of digital TV,” *IEEE Trans. Consumer Electron.*, vol.55, no.4, pp. 2242-2247, Nov. 2009.

[34] E. J. Jang, R. Woo, and D. S. Han, “Improvement of Connectivity between Infrastructure and consumer devices for infotainment services,” *IEEE Trans. Consumer Electron.,* vol. 59, no. 2, pp. 329-334, May 2013.

BIOGRAPHIES

Ricardo E. V. de S. Rosa received his M.Sc. degree in electrical engineering in 2010 from the Federal University of Amazonas. He is currently a Ph.D. candidate at Federal University of Minas Gerais. His research interests include development of applications for mobile devices, reuse based software engineering and application of computational intelligence techniques for content personalization.

Vicente F. de Lucena Jr. (M’94) received his Ph.D. degree (Dr.-Ing) in 2002 from the University of Stuttgart in Germany. Since 1990, he has been a Faculty Member with the Engineering College at the Federal University of Amazon (UFAM) in Manaus – Brazil. He is also with the Electronics and Information Technology R&D Center a research group that works with consumer electronics. His research interests include automation systems, new software engineering approaches, and the development of embedded systems.

Lucas C. Cordeiro received the B.Sc. degree in electrical engineering and the M.Sc. degree in informatics from the Federal University of Amazonas (UFAM), in 2005 and 2007, respectively. He received the Ph.D. degree in computer science from the University of Southampton in 2011. Since 2011 he has been an adjunct professor in the Electrical and Computer Engineering Department at UFAM. His work focuses on software verification, model checking, satisfiability modulo theories, and embedded systems.

João E. Chaves Filho received his Ph.D. degree in Electrical Engineering in 1997 from the Universidade Federal de Campina Grande (UFCG), Paraíba, Brazil. His M.Sc. was also obtained from UFCG in 1991. Since 1980, he has been a Faculty Member with the Electrical Engineering Department at UFAM. His research interests include new industrial automation proposals, artificial intelligence, control systems, and system identification.