On the Automatic Configuration of Application-Oriented Operating Systems

Gustavo Fortes Tondello and Antônio Augusto Fröhlich
Federal University of Santa Catarina (UFSC)
Laboratory for Software/Hardware Integration (LISHA)
PO Box 476 - 88049-900 Florianópolis - SC - Brazil
{tondello | guto}@lisha.ufsc.br

Abstract

This paper presents an alternative to achieve automatic run-time system generation based on the Application Oriented Systems Design method. Our approach relies on a static configuration mechanism that allows the generation of optimized versions of the operating system for particular classes of applications, promoting a better utilization of available resources. The EPOS *operating system, with its strategies and tools, is taken as a case-study along the text to exemplify and corroborate the proposed ideas.*

1. Introduction

Previous studies have demonstrated that embedded and mobile application do not find adequate run-time support on ordinary all-purpose operating systems, since these systems usually incur in unnecessary overhead that directly impact application's performance [1, 7]. Each class of applications has its own requirements regarding the operating system, and they must be fulfilled accordingly.

The *Application-Oriented System Design* (AOSD) method [3] is targeted at the creation of run-time support systems for dedicated computing applications, in particular, embedded, mobile and parallel ones. An *application-oriented operating system* arise from the proper composition of selected software components that are adapted to finely fulfill the requirements of a target application. In this way, we avoid the traditional "got what you didn't ask for, yet didn't get what you needed" effect of generic operating systems. This is particularly critical for mobile embedded applications, for they must often run on platforms with severe resource restrictions (e.g. simple microcontrollers, limited amount of memory, etc).

Nonetheless, delivering each application a tailored run-time support system, besides requiring a comprehensive set of well-designed software components, also calls for sophisticated tools to select, configure, adapt and compose those components accordingly. That is, *configuration management* becomes crucial to achieve the announced customizability.

This paper approaches configuration management in application-oriented operating systems, taking the strategies and tools currently deployed in EPOS as a case-study of automatic operating system configuration for mobile applications. The following sections describe the basics of the Application-Oriented System Design method, a strategy to automatically configure component-based systems from their logical description. Subsequently, the current prototypes are discussed followed by author's conclusions.

2. Application-Oriented System Design

The idea of building run-time support systems through the aggregation of independent software components is being used, with claimed success, in a series of projects [2, 6]. However, software component engineering brings about several new issues, for instance: how to partition the problem domain so as to model really reusable software components? how to select the components from the repository that should be included on an application-specific system instance? how to configure each selected component and the system as a whole so as to approach an optimal system?

Application-Oriented System Design proposes some alternatives to proceed the engineering of a domain towards software components. In principle, an application-oriented decomposition of the problem domain can be obtained following the guidelines of *Object-Oriented Decomposition*. However, some subtle yet important differences must be considered. Variability analysis, as carried out in object-oriented decomposition, does not emphasizes the differentiation of variations that belong to the essence of an abstraction from those that emanate from the execution scenarios

being considered for it. Abstractions that incorporate environmental dependencies have a smaller chance of being reused in new scenarios, and, given that an application-oriented operating system will be confronted with a new scenario virtually every time a new application is defined, allowing such dependencies could severely hamper the system.

Nevertheless, one can reduce such dependencies by applying the key concept of *Aspect-Oriented Programming* [4], i.e. aspect separation, to the decomposition process. By doing so, one can tell variations that will shape new family members from those that will yield scenario aspects. For example, instead of modeling a new member for a family of communication mechanisms that is able to operate in the presence of multiple threads, one could model multithreading as a scenario aspect that, when activated, would lock the communication mechanism (or some of its operations) in a critical section.

Based on these premises, Application-Oriented Systems Design guides a domain engineering procedure that models software components with the aid of three major constructs: families of scenario-independent abstractions, scenario adapters and inflated interfaces.

Families of scenario independent abstractions: during domain decomposition, abstractions are identified from domain entities and grouped in families according to their commonalities. Yet during this phase, aspect separation is used to shape scenario-independent abstractions, thus enabling them to be reused in a variety of scenarios. These abstractions are subsequently implemented to give rise to the actual software components.

Scenario adapters: as explained earlier in this article, Application-Oriented System Design dictates that scenario dependencies must be factored out as *aspects*, thus keeping abstractions scenario-independent. However, for this strategy to work, means must be provided to apply factored aspects to abstractions in a transparent way. The traditional approach to do this would be deploying an *aspect weaver*, though the *scenario adapter* constructhas the same potentialities without requiring an external tool. A scenario adapter wraps an abstraction, intermediating its communication with scenario-dependent clients to perform the necessary scenario adaptations.

Inflated interfaces: summarize the features of all members of a family, creating a unique view of the family as a "super component". It allows application programmers to write their applications based on well-know, comprehensive interfaces, postponing the decision about which member of the family shall be use until enough configuration knowledge is acquired. The binding of an inflated interface to one of the members of a family can thus be made by automatic configuration tools that identify which features of the family were used in order to choose the simplest realization that implements the requested interface subset at compile-time.

3. Software Component Configuration

An operating system designed according to the premises of Application-Oriented System Design, besides all the benefits claimed by software component engineering, has the additional advantage of being suitable for automatic generation. The concept of inflated interface enables an application-oriented operating system to be automatically generated of out of a set of software components, since inflated interfaces serve as a kind of requirement specification for the system that must be generated.

An application written based on inflated interfaces can be submitted to a tool that scans it searching for references to the interfaces, thus rendering the features of each family that are necessary to support the application at run-time. This task is accomplished by a tool, the `analyzer`, that output an specification of requirements in the form of partial component interface declarations, including methods, types and constants that were used by the application.

The primary specification produced by the `analyzer` is subsequently fed into a second tool, the `configurator`, that consults a build-up database to further refine the specification. This database holds information about each component in the repository, as well as dependencies and composition rules that are used by the `configurator` to build a dependency three. Additionally, each component in the repository is tagged with a "cost" estimation, so that the `configurator` will chose the "cheapest" option whenever two or more components satisfy a dependency. The output of the `configurator` consists of a set of keys that define the binding of inflated interfaces to abstractions and activate the scenario aspects and configurable features eventually identified as necessary to satisfy the constraints dictated by the target application or by the configured execution scenario.

The last step in the generation process is accomplished by the `generator`. This tool translates the keys produced by the `configurator` into parameters for a statically metaprogramed component framework and causes the compilation a tailored system instance.

4. Software Component Description

The strategy used to describe components in a repository and their dependencies plays a key role in making the just described configuration process possible. The description of components must contain enough information so that the `configurator` will be able to automatically identify which abstractions better satisfy the requirements of the application, and this without generating conflicts or invalid configurations and compositions.

The strategy to describe components proposed here could indeed be taken further as to specify components, for it encompasses much of the information needed to implement components, including their interfaces and relationships to other components. It is based on a declarative language implemented around the *Extensible Markup Language* (XML) and target at the description of individual families of abstractions[1]. The most significant elements in the language will be explained next, taking as basis the corresponding *Document Type Definition*) (DTD) fragments.

4.1. Families of abstractions

The declaration of a family of abstractions in our language consists of the family's inflated interface, an optional set of dependencies, and optional set of traits, its common package and a set of family members (software components), like this:

```
<!ELEMENT family (interface, common, member
    +, (feature, dependency, trait)* )  >
```

The inflated interface of a family, as explained earlier, summarizes the features of the whole family, while the common package of a family holds type and constant declarations that are common to all family members.

The `member` element shown bellow is used to describe each of the members in a family. It is at the heart of the automatic configuration process, enabling tools to make the proper selection while looking for inflated interface realizations. A family member is declared as:

```
<!ELEMENT member (super, type, constant,
    constructor, method, trait, cost,
    feature, dependency) * >
```

The `super` element enables a member to inherit declarations of other members in the family.Elements `type`, `constant`, `constructor` and `method` describe the interface of the member. A member's interface designates a total or partial realization of the family's inflated interface. Element `trait`, which can also be specified for the family as whole, designates a configurable feature that can be set by users, via configuration tools, in order to influence the instantiation of a component A trait of a component can also be used to specify configuration parameters that cannot be automatically figured out, such as the amount of memory available on the target platform.

Additionally, each member of a family is tagged with a relative cost estimation that is used by the configuration tools in case multiple members satisfy the constraints to realize the family's inflated interface in a given execution scenario. This cost estimation is currently rather simplistic, consisting basically of an overhead estimation made by the component developer. More sophisticate cost models, including feed-back from the configuration tools, are planed for the future.

4.2. Dependencies

The description of the interfaces in a family of abstractions is the main source of information for the proposed configuration tools, but correctly assembling a component-based system goes far beyond the verification of syntactic interface conformance: non-functional and behavioral properties must also be conveyed. For this purpose, our component description language includes two special elements: `feature` and `dependency`. These elements can be applied to virtually any other element in the language to specify features provided by components and dependence among components that cannot be directly deduced from their interfaces. Enriching the description of components with features and dependencies can significantly improve the correctness of the assembly process, helping to avoid inconsistent component arrangements.

We have chosen a feature-based model to describe dependencies among components. To satisfy the dependencies, each family implements a feature with the same name of the family, and each family's member implements a feature with its same name. Families or members can also implement aditional features using the `feature` element. For instance, consider a family of wireless network abstractions. Some members could declare a "reliable" feature, making them eligible to support an application whose execution scenario demands for reliable communication. Similarly, members of a family of communication protocols could specify the dependency on a "reliable" wireless network infrastructure, while other could implement the feature themselves.

A feature has a name and, optionally, a value. The name should be regarded as a meaningful feature in the application domain. Considering the example above, we could specify the reliable feature of a wireless network as follows:

```
<family name="Wireless_Network">
  <interface>...</interface>
```

[1] A complete description of the software component repository is obtained simply by merging individual families' descriptions.

```
  <member name="Wi-Fi">
    <feature name="reliable" />
  </member>
</family >
```

and the dependency in the protocol family as:

```
<family name="Wireless_Protocol">
  <interface>...</interface>
  <dependency feature="Wireless_Network" />
  <member name="Active_Message">
    <dependency feature="reliable" />
  </member>
</family >
```

It is important to mention that the fact of the `Active_Message` member of the `Wireless_Protocol` family requiring a reliable `Wireless_Network` does not summarily excludes members that do not implement this feature: the `configurator` would first check whether a scenario aspect is available that could be applied to a non-reliable network in order to make it behave as a reliable one.

5. Supporting Tools

At the present, we have prototype implementations of the `analyzer` for applications written in C++ and JAVA. These tools are able to parse an input program and produce a list of the system abstraction interfaces (inflated or not) used by the program, identifying which methods have been invoked and, in the case of JAVA, in which scope they have been invoked.

This information serves as input for the `configurator`, which is currently being developed. The CONFIGURATOR is indeed implemented by two tools. The first one is responsible for executing the algorithm that will select which members of each family will be included in the customized version of the system. This algorithm consists in reading the requirements found by the `analyzer` and compare them with the interfaces of each member of the family as specified in the repository. Every time a new member is selected, its requirements are recursively verified, including in the configuration any members from other families that are needed to satisfy them. The second part of the `configurator` is a graphical tool that allows the user to browse an automatically generated configuration, making manual adjustments, if needed. Moreover, the user will have to enter some important information not discovered automatically: the configuration of the target machine (architecture, processor, memory, etc.) and the values of the traits of each component.

At last, the configuration keys outputted by the `configurator` are used by the `generator`, which is implemented as a wrapper for the *GNU Compiler Collection*, to compile the system and generate a boottable image.

6. Conclusion

In this article we have presented an alternative to achieve automatic run-time system generation taking as base a collection of software components developed according with the Application-Oriented System Design methodology. The proposed alternative consists of a novel component description language and a set of configuration tools that are able to automatically select and configure components to assembly an application-oriented run-time support system.

The described configuration tools are in the final phase of development and allows the exposition of the system libraries to application programmers through a repository of reusable components described by their inflated interfaces, which are automatically bound to a specific realization at compile time. This is possible due to the component specification model that contains all the information needed to generate valid and optimized configurations for each application. This architecture makes possible the creation of versions of the system optimized for the target applications, assuring that the performance levels and resource usage optimization will be within the levels accepted by mobile applications.

References

[1] Thomas Anderson. The Case for Application-Specific Operating Systems. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.

[2] Bryan Ford et al. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM SOSP*, October 1997.

[3] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series, August 2001.

[4] Gregor Kiczales et al. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, June 1997.

[5] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[6] Alastair Reid et al. Knit: Component Composition for Systems Software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[7] Friedrich Schön et al. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP Workshop on Distributed and Parallel Embedded Systems*, October 1998.