Using software components to automatically configure Application-Oriented Operating Systems

Gustavo Fortes Tondello and Antônio Augusto Fröhlich
Laboratory for Software/Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
PO Box 476 - 88049-900 Florianópolis - SC - Brazil
E-mail: {tondello | guto}@lisha.ufsc.br
Homepage: http://epos.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 each of the applications that are going to use it. This strategy is of great value in the domain of high performance computing since it results in performance gains and resource usage optimization.

Keywords: application-oriented operating systems, software components, configuration management.

1 Introduction

Previous studies have demonstrated that embedded and high performance applications 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, 12]. 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 [6] is targeted at the creation of run-time support systems for dedicated high performance 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 high performance embedded applications, for they must often be executed 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 a 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 embedded and parallel applications. The following sections describe the basics of the Application-Oriented System Design method, a strategy to automatically configure component-based systems and a strategy to describe the components for that purpose. Subsequently, the current prototypes are discussed along with a real example of the configuration process, followed by an outline of the next steps planed for the project along with 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 [3, 5, 11, 2]. 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* [4]. However, some subtle yet important differences must be considered. First, object-oriented decomposition gathers objects with similar behavior in class hierarchies by applying variability analysis to identify how one entity specializes the other. Besides leading to the famous "fragile base class" problem [9], this policy assumes that specializations of an abstraction (i.e. *subclasses*) are only deployed in presence of their more generic versions (i.e. *superclasses*).

Applying variability analysis in the sense of *Family-Based Design* [10] to produce independently deployable abstractions, modeled as members of a family, can avoid this restriction and improve on application-orientation. Certainly, some family members will still be modeled as specializations of others, but this is no longer an imperative rule.

A second important difference between application-oriented and object-oriented decomposition concerns environmental dependencies. 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* [8], 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.

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*, identified from domain entities and grouped 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* [7], used to apply factored aspects to abstractions in a transparent way with the same potentialities of the traditional approach to do this by deploying an *aspect weaver*, but 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*, that 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 used 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 create the description of the system's configuration. 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 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 of a tailored system instance. An overview of the whole procedure is depicted in Figure 1.

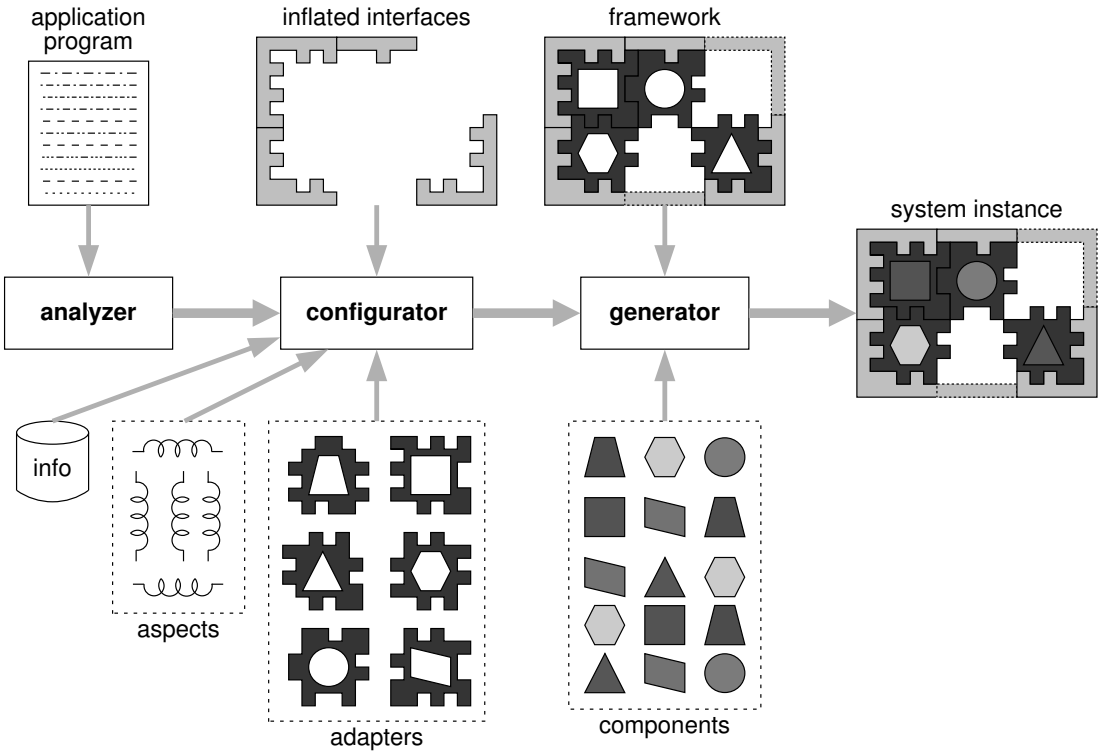

Figure 1: An overview of the tools involved in automatic system generation.

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 be complete enough 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 description declarative language implemented around the *Extensible Markup Language* (XML) [13] and target at the description of individual families of abstractions[1]. The most significant elements in the language will be explained next.

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). A fragment of a family description could look like this:

```
<family name="Address_Space">
  <interface>
    <method name="attach" return="Log_Addr">...</method>
  ...</interface>
  <common>...</common>
  <member name="Flat_AS" cost="1">
    <method name="attach" return="Log_Addr">...</method>
  ...</member>
  <member name="Paged_AS" cost="5">
  ...</member>
</family>
```

4.2 Dependencies

Although we can use the `analyser` tool to discover the dependencies of the application regarding the families' interfaces, this tool cannot be used to discover the dependencies that a family's implementation has on another one. Thus, this kind of depency must be explicited by the programmer through the `dependency` member. This dependency may happen to the entire family or just on individual members.

We have choosen to use 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:

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

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

and the dependency in the protocol family as:

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

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 verifies that all requisits found by the `analyser` were satisfied as well as dependencies among components. 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, as the configuration of the target machine (architecture, processor, memory, etc.), for instance.

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 allow 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 uses component-based strategies to generate optimized versions of the operating system for the target applications, assuring that performance levels and resource usage for embedded and parallel aplications will be certainly better that those achieved with general-purpose operating systems.

References

[1] Thomas Anderson. The Case for Application-Specific Operating Systems. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 92–94, Key Biscayne, U.S.A., April 1992.

[2] Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Germany, June 1999.

[3] Danilo Beuche, A. Guerrouat, H. Papajewski, Wolfgang Schrder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, St Malo, France, May 1999.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[6] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[7] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväkylä Finland, June 1997. Springer.

[9] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer.

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

[11] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component Composition for Systems Software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, U.S.A., October 2000.

[12] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, October 1998.

[13] World Wide Web Consortium. *XML 1.0 Recommendation*, online edition, February 1998. [http://www.w3c.org].