

BOTS: A Constraint-based Component System for Synthesizing Scalable Software Systems *

Raju Pandey and Jeffrey Wu

Department of Computer Science
University of California, Davis
{pandey,wujt}@cs.ucdavis.edu

Abstract

Embedded application developers create applications for a wide range of devices with different resource constraints. Developers want to maximize the use of the limited resources available on the device while still not exceeding the capabilities of the device. To do this, the developer must be able to scale his software for different platforms. In this paper, we present a software engineering methodology that automatically scales software to different platforms. We intend to have the application developer write high level functional specifications of his software and have tools that automatically scale the underlying runtime. These tools will use the functional and non-functional constraints of both the hardware and client application to produce an appropriate runtime. Our initial results show that the proposed approach can scale operating systems and virtual machines that satisfy the constraints of varying hardware/application combinations.

Categories and Subject Descriptors D [2]: 2

General Terms Design, Performance, Languages

Keywords Embedded Systems, Runtime Systems, Components, Constraints, Wireless Sensor Networks, Generative Programming

1. Introduction

Embedded systems are increasingly becoming an important target for developers. From PDAs to automobile break controllers to tiny wireless sensor platforms, embedded controllers allow manufacturers to embed increased functionality into everyday devices. The wide range of uses and conditions that exist for these embedded devices has led to an incredible amount of heterogeneity among these devices. Devices differ within their functional and resource (non-functional) capacities. Some devices have significant amounts of memory, a fast CPU, large internal and external storage devices and operate much like desktop systems. Others (such as wireless sensor node platforms and Java cards) have much slower processors, tiny amounts of SRAM (in the 4K-10K range), smaller storage areas, slower wireless radios and small amounts of battery power.

* This work is supported in part by NSF grants CNS-0435531, CNS-0520269, and EIA-0224469.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LC TES'06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-362-X/06/0006...\$5.00.

This paper addresses the key issues in developing and building software systems that can run on a wide range of devices.

One approach is to custom develop a software system for each device class. Currently, this approach is used in many application and system software developments. For instance, many versions of the Java virtual machines are custom developed to meet the constraints of the specific embedded devices. Similarly, several distributed wireless sensor network applications [29, 35] are developed by building separate programs for specific embedded devices: one set of programs for sensor nodes, another set of programs for gateways, and yet another for the base station, each possibly written in a different language (nesC for sensor nodes and Java/C/C# for gateways and base stations). Clearly, building custom made software is too cumbersome and expensive, and does not effectively reuse common parts of the applications. Such an approach does not adapt well to cases when new device types are added or when the functionality of the devices changes.

Another approach is to provide high level abstractions (such as APIs, and virtual machines), and port those abstractions across different devices. However, portability works great in systems that have similar sets of capabilities and resources. Portability only addresses the functional heterogeneity problem; it does not address the extreme variations in embedded system platforms. For example, consider the Java virtual machine (JVM), which acts as a portable software substrate for Java applications. While the notion of the JVM as a portability layer has worked great for desktop systems where Java applications can run on a wide variety of these systems, porting JVM on embedded devices have required device-specific JVM implementations that impose restrictions on (i) the Java language abstractions that can be used, (ii) number of objects that can be dynamically constructed, and (iii) specific runtime services (such as threading and garbage collection) that are supported. This has led to development of applications that are VM specific, and hence device specific. In addition, while the device-specific implementations of system software (operating systems and virtual machines) exploit the characteristics of the device, applications cannot transparently exploit the capabilities and resources of the underlying device. Writing an application within the embedded systems space is not simply a question of making everything bigger or everything smaller. It involves making tradeoffs that meet application requirements and embedded device capabilities.

What is needed is a software development methodology and tools that will make it easier to develop software systems that can be easily adapted to run on devices with wide range of capabilities. Such a methodology will:

- allow developers to focus on high level abstractions (such as API and VMs), thereby hiding the functional heterogeneity among different devices,

- allow developers to express tradeoffs among different design choices that depend on capabilities and resources of devices,
- allow developers to manage different resource constraints independently,
- allow developers to capture both device-independent and device-specific choices,
- allow developers to extend their current design as new devices with new capabilities are added, and
- automate the task of building device-specific software systems from a common implementation.

This paper presents the notion of *scalable* software development for a wide variety of devices. The idea of scalable software development is based on the insight that components of a software system can be divided into two classes: (i) *common* components that are independent of specific device characteristics, and (ii) *variants* that denote specific design points and that are parameterized by different design choices, device capabilities, and resources. Building a software system involves selecting common and variant components that match user preferences, software system requirements and device constraints. To illustrate software scaling, consider the example of selecting a garbage collector for a language runtime environment. An important component of a garbage collector is how objects are stored in memory: there are several implementations possible, each with different memory and time overhead. For each possible object layout, many different implementations of object compaction algorithms exist, each parameterized by space, time and functionality. Building a scalable garbage collector will involve representing all common and variant components. Selecting a specific garbage collector for a specific device will involve selecting a garbage collector component that meets the resource limitations of a specific platform: slow but space efficient for tiny sensor nodes (such as Mica [24] or Telos [36]); efficient with high space overhead for gateway nodes (such as XYZ [31] and Stargate [7]).

Developing a framework for building scalable software systems requires addressing several issues: The first is the notion of scalability and how it manifests itself in different aspects of software development. In Section 2, we describe the overall framework for a scalable software development methodology within a generative programming framework. We also identify the different kinds of design choices that software developer may come across when building scalable software systems, and how they can be represented. The second issue is the nature of software infrastructure needed to build scalable software systems. In Section 3, we describe a component-based programming language, BOTS, that we have designed specifically for capturing complex relationships among different components of a software system. BOTS can be used to represent all layers in a software stack, including operating systems, virtual machines, middleware and applications. The language supports a fine-grained composition mechanism, which allows software developers to express design choices that are not only global in nature, but that also represent local design tradeoffs. Such a representation of software systems enables component selection at both coarse and fine-grained levels. We also describe a tool for automating the generation of a software system for specific embedded devices. The tool uses a constraint solver to determine a set of common and variant components that meet application requirements, user preferences, and embedded device constraints. The tool also generates any glue code needed to connect binaries for different embedded system platforms.

We have used the methodology and a tool to build an embedded software stack, including WSN applications, middleware, virtual machine and low level operating system. In Section 4, we describe how the methodology is used to automatically generate software

substrates for a wide variety of WSN embedded devices. We also analyze the performance characteristics of the BOTS tool in this section. Section 5 describes related work. Section 6 contains conclusions and a summary of future work.

2. Scalable Software Stacks for Embedded Systems

In this section, we first describe the notion of scalable software development. We then briefly describe the overall framework, which forms the basis for designing and implementing BOTS.

2.1 Scalable Software

Software scalability is the ability to make tradeoffs in response to changing demands and resource constraints. Traditionally, the notion of scaling has been used to characterize the ability to adapt to changes in demand or resources at runtime. However, we define scalable software development as one that provides a developer the ability to easily make tradeoffs when retargeting his application *at build time* to a wide range of new platforms and demands.

Software performance is a function of three factors: the performance properties of the hardware platform, the input, and the software algorithm being used. In scaling software, a developer takes these three factors into account to choose his algorithm. We should note that “performance” refers not only to the speed of the software, but other resource usage properties such as power consumption and memory usage. For example, take the example of a wireless sensor network (WSN) application with a network topology cache. Sensor nodes often have a small amount of SRAM memory and a larger amount of flash memory. The mobility of the nodes (input) and the size of the memory (hardware) would be the factors in deciding whether to store the cache in flash or SRAM (algorithm). If the nodes are mobile (the cache must update positions regularly) and there is sufficient SRAM memory, then SRAM would be the logical place since Flash has a high energy cost. If the SRAM size is small, then Flash is the only choice. If the nodes are stationary, then write costs may not matter and Flash would be the logical choice regardless of the SRAM size. Of course, this decision must be balanced with the memory requirements of the rest of the application.

Software scaling requires the developer to understand the functional nature of the code he is using – along with the performance implications of the code on the platform he targets and the input he uses. Functional properties are well understood and often well documented, since developers must fully comprehend the functional properties of the code which he includes into his application. Functional properties are often expressed in such a formal way that automated tools can verify their correctness (type checkers, last-use analyzers, etc). We do not have the same level of tools for non-functional properties. Input and hardware characteristics often affect the non-functional properties of code. With the numerous combinations of applications and hardware, the array of non-functional properties is impossible to manually manage.

Understanding the non-functional properties of code may not be a problem when a developer exclusively uses his own hand-written code. However, as he wants to scale his application to an increasing set of different devices, the developer needs methods of reusing code from third parties. While information about the functional properties of the third party code is usually transferred in API documentation, we currently have insufficient methods of formally encoding the non-functional properties. This leaves the consuming developer to discover the non-functional properties through his own tests or to re-implement the functionality so that he knows the properties of the code he is working with. We make a case for codifying these tradeoffs so that optimized software stack for embedded

applications can be automatically generated. This would allow the developer to work increasingly at the platform-independent level.

2.2 A Framework for Scalable Software

The framework for building scalable software is based on the notion of generative programming. Czarnecki and Eisenecker [10] define generative programming as follows: “Generative programming is a software engineering paradigm based on modeling of software system families such that, given a particular requirements specifications, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.” This definition emphasizes several key ideas: (i) *system families*, (ii) *reusable implementation components*, (iii) *configuration knowledge*, and (iv) *product requirements*. Unlike the traditional software engineering methodology, the focus in generative programming is on developing a family of software products. It involves modeling a product family in terms of (i) a set of features [25] that are common across all members of the family and (ii) a set of variant features that differentiate among the family members. A product family is implemented in terms of reusable components, which implement common and variant features using different implementation techniques (such as components, generic and parameterized programming [20], aspect-oriented programming [26], generators [1] and meta-programming). The configuration knowledge describes how implementation components can be composed together to build more complex components. Product requirements drive the construction of a final product by using the configuration knowledge for selection, transformation and composition of implementation components.

We use the notion of the software product family to denote a high level view of a software stack that will run on a wide variety of embedded devices. Development of a software system, thus, involves identifying common and variant components. The variant components require identifying design tradeoffs of three aspects of software artifacts:

- **States** capture specific knowledge in a computation. They can be stored at many different levels. The decision to choose a specific state will be influenced by the context in which system operates. Consider the case in which a sensor device caches knowledge about its neighbors. Large devices (such as gateways) may cache the entire routing tree, whereas smaller devices may only store neighbor information.
- **Algorithms** transform knowledge. The choice of algorithms for a computation will depend on several factors, including CPU speed, memory resources, application requirements (fidelity, quality, etc.), and user preferences.
- **Information representation** of an abstraction captures details regarding the abstraction. Consider the binary representation (such as JVM byte code or ELF) of programs. Such information can be represented at different levels of details: some may include all semantic information (such as types, scope, names, etc.); other may strip away all symbolic information to save space. The level of detail at which specific abstractions can be implemented depends on the context in which a specific application will be deployed and run.

Given the definition of the common and variant parts, building a software system for a specific device involves identifying common and variant parts that meet specific requirements. Below we describe the BOTS system for representing product families, and for generating specific product for a specific platform.

```
interface object_layout {
    method Object new_object(int size);
    method byte [] get_object();
    method void set_object(byte [] obj);
    attribute int overhead;
    attribute string latency = ‘‘medium’’;
}
interface garbage_collector {
    method void start();
    method void gc();
    attribute int overhead = 4;
}
```

Listing 1. Interface for the garbage collector and object layout controller

3. The BOTS System

We now describe the BOTS system for representing and building scalable software systems. There are two aspects of the BOTS system: one is the language used for capturing both common and variant parts of a software system. The second is the mechanism to select specific parts that will meet user requirements. BOTS takes a database of software components and produces a component arrangement or a *configuration*. The configuration is limited by the constraints of hardware properties, input descriptions, and developer provided direction. BOTS creates a component configuration and reifies this abstract configuration into a *makefile* for the developer to run. By influencing BOTS’s choices using constraints, the developer can create a scaled version of his software.

3.1 The BOTS component language

The BOTS component language provides a component-based framework for representing embedded applications and system software (the virtual machine, the operating system, and middleware). The design of the language has been driven by two requirements: First, the framework must allow for the coupling of functional and non-functional properties of hardware devices and software artifacts so that decision about both component selection and scaling decisions can be automated. Second, given that a software stack on a device will include components implemented in several programming languages (such as assembly, C and Java), the framework must support the ability to represent components described and implemented in different language. In other words, the component language must support separation of the structure, composition, and scalability aspects of a system from its actual implementation. Below we briefly describe the language that has been driven by these concerns.

The BOTS language is based on four key ideas: *interfaces*, *components*, *attributes*, and *constraints*.

3.1.1 Interfaces

Interfaces encapsulate the service-oriented notion of software entities. In our system, interfaces are signatures of the functional and non-functional properties of components. BOTS interfaces are similar to OO interfaces, which describe the signatures of concrete classes.

In Listing 1, we show two example interface components: *object_layout* and *garbage_collector*. The *object_layout* component provides an interface for implementing layout of Java objects in memory. There are many possible implementations of the abstraction, each of which is characterized by two attributes: *overhead* and *latency*.

```

component group_layout implements
  object_layout {
    attribute overhead = .75;
    // latency is the time needed to find
    // free locations for new objects
    attribute latency = 'high';
    // component defined in a c file
    artifact C { File: "glayout.c" }
  }
component random_layout implements
  object_layout {
    attribute overhead = 8;
    attribute latency = 'low'; ...
  }
component class_layout implements
  object_layout {
    attribute overhead = 4;
    attribute latency = 'medium'; ...
  }
component regional_collector implements
  garbage_collector {
    import object_layout;
    attribute overhead = object_layout.
      overhead + 4; ...
  }
component reference_collector implements
  garbage_collector {
    import object_layout;
    attribute overhead = object_layout.
      overhead + 4; ...
  }
component mark_and_sweep implements
  garbage_collector {
    import object_layout;
    attribute overhead = object_layout.
      overhead +4; ...
  }

```

Listing 2. Components implementing `object_layout` and `garbage_collector`

3.1.2 Components

A component is an atomic collection of *software artifacts* and non-functional *attributes* that describe some of the runtime properties of the artifacts. Components implement services described in interfaces. In addition, components may express dependency relationships between themselves and other services. In Listing 2, we show components that characterize different implementations of the `garbage_collector` interface.

Each implementation binds specific values of attributes either directly or in terms of attributes of other components. For instance, the `overhead` attribute of `regional_collector` depends on the `overhead` attribute of the `object_layout` component. In Figure 1, we show the relationship among interfaces, components, and attributes.

Artifacts Components represent collections of software artifacts such as files, methods, variables, types, scripts, etc. BOTS implements a general component framework that allows developers to represent artifacts from C, Java, Python, NesC or any programming language one wishes to use. Different types of constructs are managed by plugin *namespace managers*. For example, a C namespace manager will handle the C functions and variables while a Java namespace manager handles Java classes. If a developer needs el-

ements not currently managed by one of our plugins, he can add his own. Elements are generated from *artifact* sections, like the one seen in Listing 2. The artifact section is passed to a namespace manager, which generates elements that become part of the enclosing component. In the example, the C namespace manager parses the `glayout.c` file and adds the methods and variables from that file to the `group_layout` component.

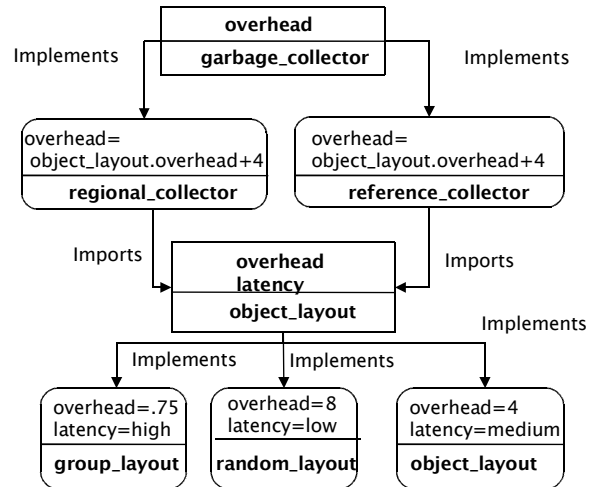


Figure 1. Parameterized relationship between Interfaces and Components

Component Dependencies A component may depend on another component for providing specific services. In Listing 3, we show the dependencies between the `Radio` and the `Leds`, `Timer`, `Router` and `Listener` services.

Components in BOTS do not depend directly on other components; they depend on interfaces instead. Depending on an interface instead of an implementation adds one level of abstraction to composing components configurations. In most component systems, a component names the component it is dependent on. This means that a component not only states what services it needs, but also how those services are to be implemented. In BOTS, a component states what services it needs by depending on an interface instead of components. For example, instead of a VM component depending on a thread management component, the VM component would depend on a thread management interface. Multiple components may implement this interface. Since components depend on interfaces, component writer to express that a component is dependent on a service without explicitly picking a specific implementation.

Therefore, if multiple components implement a given service, there are many possible legal configurations of components. How does BOTS know which configuration to pick? In previous work, the developer manually generates the *configurations* and explicitly picks the implementations [18, 21, 38]. In BOTS, a set of *constraints* are used to restrict the set of legal configurations. A constraint may require the component implementing the cache component to store its data in SRAM memory. BOTS will therefore ignore configurations where the cache stores its data in flash. We describe this process in Section 3.2.

Conditional Dependencies Both constrained and unconstrained dependencies may take the form of a conditional dependency. A conditional dependency is a dependency that may be “turned on” or “turned off” by a boolean expression. The condition is attached to the end of the dependency in the form of the keyword `if` followed by a boolean expression. Before evaluating the dependency,

```

interface Communications { ... }
interface Leds { ... }
interface Router { ... }
interface Listener { ... }
interface PacketCache { ... }
interface Timer {...}
component Radio implements Communications
{
  artifact C {File: "Radio.c"}
  import component Leds;
  import component Timer if hasTimer;
  import component Router suchthat
    component.packetSize < 15;
  import component Listener suchthat
    minimize component.codesizeBytes;

  attribute amps = 14;
}

```

Listing 3. A component description for a Radio Manager

```

component opcodeSwitchTable implements
  InstructionDispatch {
  attribute architecture = "mica2";
  attribute codesizeBytes =
    out.usedInstructions() * 15 + 500;
}

```

Listing 4. A component description for a Java VM instruction dispatcher

BOTS checks the boolean expression. If the expression evaluates to `false`, the dependency is ignored as if it never existed. The second dependency in Listing 3 is conditional. The conditional dependency states that the `Radio` component depends on the `Timer` interface only if `hasTimer` is true.

3.1.3 Attributes

Components describe their non-functional properties through attributes. The attributes describe properties such as running time, packet size, or space usage. Given an interface, different components may implement the same component with different attribute values. Through attributes, we can capture variations among component implementations. Attributes allow us to capture non-functional properties of components. Attributes may take on a string or integer value¹. Simple attributes are assignments of name-value pairs, as seen in Listing 4. Attributes need not be statically assigned values. Attributes may also be calculated through expressions, even using outside output. In Listing 4, attribute `codesizeBytes` is set to the total size of the bytecode switching table. This is calculated by counting the number of instructions (using user-defined method `out.usedInstructions()`), multiplying by 15, and adding 500 bytes for overhead.

Synthesized attributes: BOTS allows a component to derive the value of an attribute from the attributes of dependencies. These attributes, which depend on the way a component's dependencies are satisfied, are called *synthesized attributes*. To create a synthesized attribute, one needs to be able to name the components that are used to satisfy a dependency. BOTS allows the component description writer to name dependencies through labels as shown in Listing 5.

```

component AddInstruction {
  attribute speed = 2 * Storage.fetchSpeed
    + Storage.writeSpeed;

  Storage: import component Memory;
}

```

Listing 5. Example using synthesized attributes

The label is the identifier preceding the colon before the keyword `import`. Within an attribute expression, using the name `Storage` will refer to the component used to satisfy the dependency. Just as the `component.attribute` syntax allows the description writer to refer to attributes of components which are candidates for satisfying a dependency, the `Storage.fetchSpeed` will refer to the `fetchSpeed` attribute of whatever component is used to satisfy the dependency labeled by `Storage`.

3.2 Constraints

BOTS allows the programmer to constrain the possible configurations based on the non-functional properties (attributes) of the components. For example, a programmer may require that a program image not take up more than 25 of the flash memory (the attribute here being size).

Constraints are classified in two ways. First, we may classify them by their scope - constraints may be either *global* or *local*. Local constraints apply to a dependency between two components. For example, a network component requires a caching service, and may impose the constraint that cache must be at least 16KB large. The size requirement is a local constraint, since it only applies to one dependency. A global constraint may apply to the entire configuration. An example of a global constraint is that the total size of the executable may be no larger than the program memory. Second, a constraint may either be a *boolean* constraint or a *preference* constraint. Boolean constraints divide configurations into a legal set and an illegal set based on the result of a boolean expression. The configurations in the illegal set are ignored. Preference constraints order the configurations into a *preference ordering*; The configuration that best satisfies all of the preference constraints the best is the *most preferred configuration*. Constraints, thus, allow us to select specific product from a family of product descriptions.

BOTS allows the user to build constraint expressions that include hardware parameters and input properties. BOTS takes in a *hardware description file*, which characterize specific embedded devices. It then binds the hardware parameters to names. These names can then be used in constraints. Users may also use `out` methods to read the client application and describe its properties. For example, in Listing 4, we use an `out` method to read the number of used opcodes in our client Java program. This functionality allows the user to easily make the constraints relevant to a specific combination of hardware and client application for retargeting and scaling purposes

3.2.1 Local constraints

We show an example of a component with multiple dependencies - some of which are constrained dependencies - in Listing 3. The `Radio` component has five dependencies, marked by the `import` keyword. The first dependency states that this component depends on the `Leds` interface. It is an unconstrained dependency. Therefore, BOTS is free to choose any of the components in its database that export the `Leds` interface to satisfy the first dependency.

The second dependency in Listing 3 states that `Radio` requires the `Router` interface to be satisfied. However, the second `import` is accompanied by a dependency constraint, marked by the `suchthat`

¹We are currently investigating additional types for attributes

```
Interpreter.size > 1024 => Interpreter.  
location = "Flash"
```

Listing 6. A global constraint requiring that an interpreter with a size larger than 1024 bytes be placed in flash memory

```
sum ProtocolHandler.size < 1024
```

Listing 7. A global constraint requiring that all of the protocol handling code fit into 1KB

```
min Interpreter.size
```

Listing 8. A global constraint minimizing the size of the interpreter

clause. The `suchthat` clause tells BOTS that the component that is used to satisfy this dependency must have a `packetSize` attribute less than 15. This constraint is a *binary local constraint*; it specifies a predicate that must hold true when a component is selected to implement the Router service.

The `suchthat` clause can not only be used to limit the choices, but also to specify a preferred component. The third import statement in Listing 3 uses the `minimize` keyword, which specifies that the component with the lowest `codeSizeBytes` attribute should be selected. This is a *preference local constraint*.

A `component.attribute` expression within a `suchthat` clause causes BOTS to read the attribute value of the candidate component, allowing BOTS to differentiate among components based on their non-functional properties. This allows developers to optimize for space, speed, power usage, etc. and set non-functional limits on performance.

3.2.2 Global constraints

Local constraints allow a user to control individual dependencies. However, users may want to express their constraints in terms of the entire application. For example, we may require the application constants to be stored in flash memory if there are too many of them. This is a property of the entire program, not just one single component, and is best described using *global constraints*. An example of a boolean constraint is shown in Listing 6. The constraint specifies that an interpreter larger than 1024 bytes be placed in flash. Listing 7 shows another instance of a global constraint asserting that the total size of all of the protocol handlers must not exceed 1024 bytes. Such constraints emphasize tradeoffs with respect to the space requirements. Global constraints also have a preference form, as seen in Listing 8. The constraint gives preference to those configurations with the smaller interpreter sizes.

3.3 The BOTS tool

The BOTS component language provides the necessary abstractions for representing common and variant components of a software systems. Design choices that reflect different tradeoffs in state, computation and information representation are encapsulated using components, attributes, and local and global constraints. A software system represented using the BOTS programming language denotes a product family. The BOTS tool generates a specific product (binary executable) from a given product description and a set of constraints. Below we briefly describe how a developer uses BOTS.

The user invokes the BOTS tool by specifying a set of starting components and a set of constraints. After reading the component and platform descriptions, BOTS builds dependency graphs by satisfying the dependencies of the start component. For each dependency, there is a set of components that may be used to satisfy the dependency. For each component in that set, the graph is cloned and each component in the set is added to its own graph. The process of building dependency graphs then continues separately for each newly created graph.

The drilling down process described above is stopped when one of the three conditions following occurs. First, BOTS may encounter a component with no dependencies. Second, it may find a case where a component is transitively dependent on itself (circular dependency). Third, a uniqueness constraint may cause the only eligible component to be one that is already in the graph. Components may be included multiple times within a graph as long as neither circularity nor uniqueness constraints are violated.

Once the drilling down process has stopped, the constraints are acted upon. As we described earlier, there are two types of local constraints: constraints that eliminate possible configurations (strict boolean constraint) from consideration, and constraints that create a preference ordering among configurations (preference constraint). Strict boolean constraints are used to prune away all components that do not satisfy the constraint. Preference constraints are saved and evaluated once the entire graph is built, and BOTS may compare the different configurations. Preference constraints are evaluated in the reverse order of that in which BOTS encountered the constraint. Therefore, the local preference constraints that are closest to the start component are evaluated last and have the most weight.

The configurations denote the possible product solutions. The BOTS tool then generates relevant wrappers and glue programs, make files, and runs the make files to build executables.

4. Application and Evaluation

BOTS was designed to help automatically generate a software stack for wireless sensor network applications. Our overall goal was to build a programming environment that mostly insulates WSN programmers from the vagaries of the underlying devices, and at the same time, allows application writers to exploit the characteristics of the devices. Below we briefly describe how we use the BOTS tool to achieve this.

4.1 Building a software stack for WSN

We now describe the overall process of building an execution environment for WSN applications. Applications are written in a WSN programming language, which we have designed. The language, called Gaggle, is an extension of Java. It includes high level abstractions for organizing and controlling sensor nodes in terms of groups, for implementing and managing synchronous and asynchronous node and group level interactions and for group-level information sharing. The high level languages (Gaggle and Java), the virtual machine that runs on each device, and the low level operating system that interfaces with the low level hardware form an abstract software stack. The abstract software stack denotes a software product family for WSN applications. Application writers use this abstraction to write WSN applications.

A Gaggle compiler analyzes the application and transforms it into Java classes. It also generates a product specification that characterizes the middleware components required to implement the application. The application classes are compiled using a Java compiler, which converts the application and support classes into the Java bytecode format. The bytecode program forms a high level product requirements of the runtime environment that will run on each sensor device. It contains information about JVM instruc-

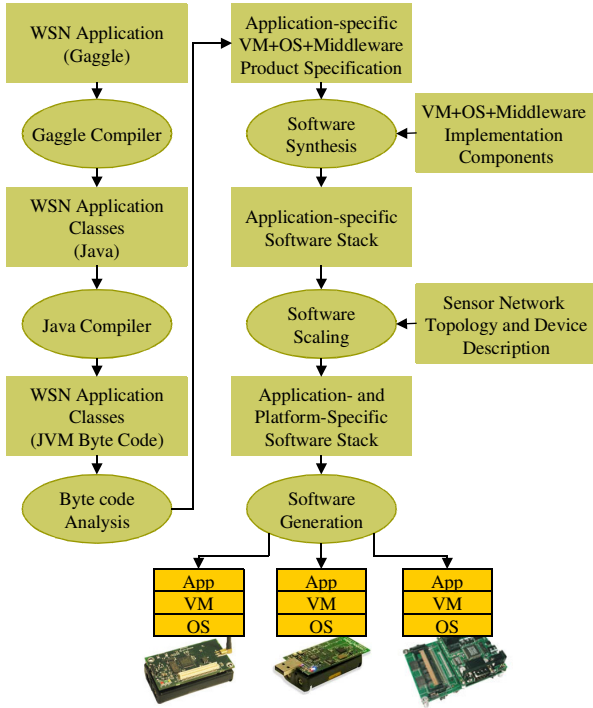


Figure 2. Generation of a runtime environment for applications

tions, language abstractions (such as type, inheritance, static/instance variables, exception handling, concurrency, synchronization, etc.), references to other Java classes or native libraries, runtime and operating system modules.

A software synthesis tool analyzes the intermediate bytecode and generates a platform-independent product specification that characterizes the VM and OS components needed to run the application on a sensor node. This specification is independent of the target platform. The high level language semantics-driven derivation of product requirements allows us to discard all components (instructions, runtime services, libraries, etc.) that are not needed to run the application. Thus, if an application only requires 20 JVM instructions to run, the product requirements for the JVM for this application will only contain references to the 20 instructions. BOTS then analyzes the platform-independent application requirements and implementations of the required VM and OS components, and *selects* those component implementations that meet the constraints imposed by the platform. Such a fine-grained composition permits construction of application-specific virtual machines and operating systems that can be run on highly resource-constrained devices, while maintaining an illusion of a complete high-level programming language for WSN application writers. The BOTS tool creates a binary for the VM, and uses the binary to patch the application class for any references to native methods. A distributed program loader loads the application and system software binary on the target device.

4.2 Evaluation

We now analyze the performance characteristics of the methodology and the BOTS tool. First, we show that we can use the design methodology to build highly resource-efficient software stacks (Sections 4.3 and 4.4). Second, we show that we can use attributes and constraints to influence selection of specific components for building the system software (Section 4.4). Finally, we show that

BOTS can produce scalable WSN software stacks that are guided by application-specific constraints (Sections 4.5 and 4.6).

For our evaluation, we have used the following applications: (i) *CntToLeds*, which displays a counter on a sensor node’s LEDs; (ii) *RfmToLeds*, which reads numbers from the radio and displays them on the LEDs; (iii) *AggServices*, which reads sensor data from neighbour nodes and aggregates them; (iv) *BirthdayBase*, which uses a birthday-paradox based probabilistic algorithm for identifying neighbor nodes; and (v) *Fibonacci* computes fibonacci numbers. All of the experiments were run by building MICA2 application binaries, which can be directly loaded on MICA2 nodes and run.

4.3 OS* synthesis

OS* is a lightweight operating system designed for resource-constrained devices. We have used the BOTS component language to capture common and variant aspects of the operating system. OS* currently runs on Mica2 and Telos sensor platforms. For the experiments here, we used the BOTS tool to build several C applications that use OS* directly.

The table below shows the sizes of the images of both the application and OS. The size of the complete OS is at the bottom for comparison.

| C Applications | Image Size | % Full OS Size |
|--------------------|------------|----------------|
| <i>CntToLeds</i> | 4KB | 12% |
| <i>RfmToLeds</i> | 13KB | 41% |
| <i>AggServices</i> | 22KB | 70% |
| <i>TreeRouting</i> | 19KB | 61% |
| Full OS | 31KB | 100% |

The result here shows BOTS’s ability to select only those components that are required by the C application.

4.4 Synthesizing a Java VM

VM* [28] is a framework for implementing virtual machines. We have implemented the JVM using this framework. VM* is written in C and is constructed from fine-grained BOTS components. Given that most Java applications use a small subset of available byte codes and Java abstractions, we can use synthesis and scaling to build a VM that only uses these abstractions.

In BOTS, we achieve byte code selection by defining an instruction dispatch component that depends on each instruction conditionally. The dispatch component is shown below:

```

component opcodesC
{
    interface opcodes;
    import all instruction_iconst_0 if out.
        hasInstruction("iconst_0");
}

```

The conditional expression is a plugin method that checks whether an instruction is used in the bytecode. Since only the dependencies for used instructions are active, BOTS only includes the implementations of the used opcodes.

The table below shows the image sizes of the applications, VM* and OS*, and the number of opcodes implemented as part of the VM. The complete OS and VM size are included at the bottom for comparison.

| Java Applications | Image Size | Number of Opcodes |
|---------------------|------------|-------------------|
| <i>CntToLeds</i> | 9KB | 20 |
| <i>Fibonacci</i> | 11KB | 10 |
| <i>BirthdayBase</i> | 20KB | 48 |
| Full OS and VM | 66KB | 201 |

```

unique interface oss_scheduler {
    attribute short size;
    attribute short max_threads;
}
interface Thread {
    attribute short count;
}

```

Listing 9. The scheduler and thread interfaces

4.5 Conditional Component Selection

VM* can store application bytecode in SRAM or Flash. Reading from SRAM is faster and less power intensive than reading from Flash memory. However, since some devices (such as MICA) have only small amounts of SRAM (around 2K-4K), application code may need to be stored in FLASH. Implementations of components that manipulate application code differ significantly depending on the location of code. We use BOTS to configure VM* using the following high level constraint:

```

if bytecode.size > 512
    then memory.type = "Flash"
    else memory.type = "SRAM"
endif

```

In this expression, `bytecode` and `memory` are two component interfaces; `bytecode` encapsulates the abstraction of application byte code, and `memory` provides access routines for accessing memory locations. BOTS uses the constraints to drive the selection of relevant components.

4.6 Scheduler selection

OS* supports multi-threading, event-based programming, and run to completion concurrency models. The different models have different overheads. Also, scheduler complexity changes depending on the form of concurrency required. For instance, multi-threaded programs require a separate stack for each thread, complex data structures for managing thread states, and support for thread context management and switching. Run to completion threads, on the other hand, require a fairly simple scheduler, which has low memory overhead and low runtime overhead. One of our goals is to use BOTS to analyze the concurrency requirements of an application and generate a scheduler that fits the needs of the application.

We have implemented a scheduler interface (`oss_scheduler`) that both a multi-threaded and a single-threaded scheduler implements. The scheduler interface mandates the presence of `size` and `max_threads` attributes. For the single-thread scheduler, `max_threads` is set to 1. For the multi-thread scheduler, `max_threads` is set to 65535. In addition, we have created a `Thread` interface that is implemented by components that generate one or more new threads. The `count` attribute is the number of threads that the component creates (several of our application components create more than one).

We first define a constraint that the application should use the smaller scheduler:

```

component app_main // main component
{
    import all oss_scheduler suchthat min
        oss_scheduler.size
}

```

We then define another constraint that restricts the usage of the single-thread scheduler to single thread applications:

```

sum Thread.count <= oss_scheduler.size

```

| Applications | Image size (Single-Threaded) | Image size (Multi-Threaded) |
|--------------|------------------------------|-----------------------------|
| CntToLeds | 2K | 4K |
| RfmToLeds | 12K | 13K |
| TreeRouting | N/A | 19K |

Table 1. Size comparison of multi-threaded scheduler vs. single threaded scheduler

| Application | Total Comp. | Included Comp. | # of Graphs | RunTime (in Sec) |
|------------------|-------------|----------------|-------------|------------------|
| CntToLeds (C) | 437 | 54 | 2 | 1.3 |
| RfmToLeds | 437 | 76 | 2 | 1.8 |
| AggServices | 441 | 88 | 2 | 2 |
| TreeRouting | 437 | 92 | 2 | 2.4 |
| BirthdayBase | 433 | 171 | 16 | 81 |
| CntToLeds (Java) | 433 | 121 | 16 | 21 |
| Fibonacci | 433 | 129 | 16 | 29 |

Table 2. Running time of BOTS on various applications

The restriction states that the total number of threads in the application must be less than or equal to the maximum number of threads that the scheduler can handle. Therefore, in cases where more than one thread is created, only the multi-threaded scheduler is valid.

Table 1 shows the results of builds that include different schedulers. There is a difference of about 2K between the single-threaded and multi-threaded versions. The tree routing application has no single threaded version and is included to demonstrate that BOTS selects the correct scheduler given the stated restrictions.

4.7 Execution profile

In this section, we show the execution time of BOTS on several applications. For each application, we list the total number of components in the component database used, the number of components that were included in the most preferred configuration, the number of graphs that were built (both legal and illegal), and the time it took for BOTS to run. The time included is the time from the invocation of BOTS to the completion of the writing of the makefiles. It does not include the application compilation process itself. Results can be seen in Table 2. Tests were run on a Dell Precision 360 with an Intel Pentium 4 2.8 GHZ processor and 1GB of RAM, running Linux kernel 2.6.12-1.1447_FC4smp.

5. Related Work

Component-based languages are increasingly being used as a basis for developing systems from reusable components. Much of the notion of interfaces, implementation and aggregate components in BOTS is based on the similar ideas in Koala [43] and Units [16]. Other examples of component-based languages include ComponentJ [40] and Jiazzi [32] for Java, and Moby [15] for ML. BOTS can be thought of as an extension to these languages, in which extensions have been added to support scalable component selection and co-product generation. Techniques such as mixins [34], subject-oriented programming [23], and feature-oriented programming [37] use some notion of separation of concerns for composing components from a set of components and aspects. The current version of BOTS does not include the notion of aspects. As we gain more understanding of the product family generation process, we will consider how BOTS can be extended to support composition of different aspects.

Component runtimes and operating systems are not new. Researchers have developed both static and dynamically generated

operating systems and virtual machines such as Choices [14], OS-Kit [17], PURE [3], 2K [13], JavaOS [39], Pebble [12], and eCos [8]. The object for many of these operating systems was to provide a platform that was easily extensible [6]. In doing so, these researchers provided a system that allowed re-targeting through the addition and removal of components. However, these systems did not have a synthesis or scaling infrastructure. The mechanism for selecting a component while building a software system is simple: component A is included if component B depends on it. There is no support for software scaling.

Many previous works have recognized the importance of reusability within the embedded software space and the potential of components to provide reusability and specialization [33, 27]. Indeed, nesC makes components and interfaces a central concept for TinyOS [18]. There have been several improvements to the nesC model: adding parameterization, more complicated linkings, and private state [21, 5]. Some of the functionality in BOTS may be replicated in nesC, particularly with a facade design pattern [19], but it still requires manual implementation by the programmer. In addition to the problems of the above component runtimes, synthesis is done at a coarse grain level. Dependencies cannot be managed at an individual level – only at the component level. Therefore, the resulting compositions may be slightly larger than necessary.

One work that is similar to ours is that of Zeller and Snelting[45]. Zeller and Snelting produced the Incremental Configuration Environment (ICE), which maintained consistent component graphs with respect to the functional requirements of each software artifact. However, expressing non-functional properties in ICE was difficult. Expressing non-functional properties of integer values was impossible. In addition, there was no easy method of creating a preference ordering of configurations. Configurations in ICE are either legal or not. This binary model does not map well to an environment where the developer is trying to express tradeoffs between different resources.

Urting et. al. created a component system that also allowed for QoS of the component [42]. However, this component system simply acted as a limiter of potential component connections; if the QoS requirements were not compatible, then the components could not work with each other. The Knit component composition system created a type system for components that could be checked to prevent illegal linkings [38]. Neither component system had the concept of a preference ordering of configurations based on the attributes.

Bhatia, Consel, and Pu [4] produced a customizable OS that uses dynamic dispatch and lazy linking in order to optimize code based on known runtime values. Our component selection model gives developers the ability to express more optimizations than can be handled by a compiler – and does so statically.

Feature models [25, 22, 41, 9, 11] and software product line engineering [44, 30] advocate designing and implementing for a product family. The feature model uses the notion of common and variant features to model a system family. The software engineering institute at CMU has developed a rigorous framework [2] of software product line practices that are being applied successfully for several commercial products.

6. Conclusion and Future Work

Device heterogeneity provides unique challenges for building embedded applications as they must run on a number of devices which differ significantly in resources, architecture, and capabilities. In addition, they need to scale to capabilities of the different platforms. In this paper, we have described a component-based system that allows developers to scale execution environments that meet the specific needs of an application, while satisfying the constraints im-

posed by the underlying platforms. Automated software scaling allows developers to adapt synthesized components further by selecting components that meet platform constraints. Our results show the effectiveness of the approach in mapping high level abstractions to highly optimized runtime environment that can be hosted on resource-constrained devices.

Our future work involves extensive evaluation of the usability of BOTS on a wide range of platforms. One of our goals is to gain insight into the process of building scalable components, how the components correlate with each other, and how they influence overall system performance. Additionally, as our collection of applications and runtime options develop, we will examine the scalability of the system and look to improve the component selection algorithm.

References

- [1] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.
- [2] J. Bergey, G. Campbell, S. Cohen, M. Fisher, B. Gallagher, L. Jones, L. Northrop, and A. Soule. Software product line acquisition: A companion to a framework for software product line practice, version 4.2. Available at http://www.sei.cmu.edu/productlines/frame_report/introduction.htm, 2005.
- [3] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schroder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. On the development of object-oriented operating systems for deeply embedded systems - the PURE project. In *ECOOOP Workshops*, pages 27–31, 1999.
- [4] Sapan Bhatia, Charles Consel, and Calton Pu. Remote customization of systems code for embedded devices. In *EMSOFT '04: Proceedings of the fourth ACM international conference on Embedded software*, pages 7–15, New York, NY, USA, 2004. ACM Press.
- [5] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proceedings of the 18th Annual ACM Symposium on Applied Computing (SAC'03)*, 2003.
- [6] W. H. Cheung and Anthony H. S. Loong. Exploring issues of operating systems structuring: from microkernel to extensible systems. *SIGOPS Oper. Syst. Rev.*, 29(4):4–16, 1995.
- [7] Crossbow Inc. *Stargate - XScale Network Interface and Single Board Computer*. <http://www.xbow.com>.
- [8] Cygnus. eCos: Embedded Cygnus Operating System. <http://www.cygnus.com/ecos>.
- [9] K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative Programming for Embedded Software: An industrial experience report. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*, number 2487 in LNCS, pages 156–172, 2002.
- [10] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
- [11] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In R. L. Nord, editor, *Proceedings of the SPLC 2004*, number 3154 in LNCS, pages 266–283, 2004.
- [12] E. Gabber et al. The Pebble component-based operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 267–282, 1999.
- [13] F. Kon et al. 2K: A reflective, component based operating system for rapidly changing environment. In *Proceedings of the European Conference on Object Oriented Programming*, 1998.
- [14] R. Cambell et. al. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, 1993.

- [15] Kathleen Fisher and John H. Reppy. The design of a class mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 37–49, 1999.
- [16] Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 236–248, New York, NY, USA, 1998. ACM Press.
- [17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.
- [18] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, 2003.
- [19] David Gay, Phil Levis, and David Culler. Software design patterns for TinyOS. In *LCES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 40–49, New York, NY, USA, 2005. ACM Press.
- [20] J. Goguen. Parameterized programming and software architecture. In *Proceedings of the fourth International Workshop on Reuse*, pages 2–11, 1996.
- [21] Benjamin Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (SNACK). In *Proceedings of the second international conference on Embedded networked sensor systems*, 2004.
- [22] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 76, Washington, DC, USA, 1998. IEEE Computer Society.
- [23] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM Press.
- [24] J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002.
- [25] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in LNCS, June 1997.
- [27] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.
- [28] J. Koshy and R. Pandey. VM*: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of the third international Conference on Embedded Networked Sensor Systems (Sensys)*, San Diego, CA, USA, Nov 2005. ACM.
- [29] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 64–75, New York, NY, USA, 2005. ACM Press.
- [30] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In C. Gacek, editor, *Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7)*, number 2319 in LNCS, pages 62–77, Austin, USA, April 2002. Springer Verlag.
- [31] D. Lymberopoulos and A. Savvides. XYZ: A Motion-Enabled, Power Aware Sensor Node Platform for Distributed Sensor Network Applications. In *Proceedings of IPSN 05, Los Angeles*, 2005.
- [32] S. McDirmid, M. Flatt, and W. Hsieh. Jiazz: New-age components for old-fashioned Java. In *Proceedings of OOPSLA*, Oct 2001.
- [33] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation*, page 200, 1994.
- [34] David A. Moon. Object-oriented programming with flavors. In *OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8, New York, NY, USA, 1986. ACM Press.
- [35] D. Ganesan P. Desnoyers and P. Shenoy. Tsar: A two tier storage architecture using interval skip graphs. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 39–50. ACM, November 2005.
- [36] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, 2005.
- [37] Christian Prehofer. Feature-oriented programming: A fresh look at objects. *Lecture Notes in Computer Science*, 1241:419–443, 1997.
- [38] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.
- [39] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System*. Addison Wesley, 1999.
- [40] Joao Costa Seco and Luis Caires. A basic model of typed components. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, London, UK, 2000. Springer-Verlag.
- [41] Janos Sztipanovits and Gabor Karsai. Generative programming for embedded systems. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 32–49, London, UK, 2002. Springer-Verlag.
- [42] David Urting, Yolande Berbers, Stefan Van Baelen, Tom Holvoet, Yves Vandewoude, and Peter Rigole. A tool for component based design of embedded software. In *CRPITS '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 159–168, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.
- [43] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.
- [44] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [45] Andreas Zeller and Gregor Snelling. Unified versioning through feature logic. *ACM Trans. Softw. Eng. Methodol.*, 6(4):398–441, 1997.