International Journal of Software Engineering
and Knowledge Engineering
Vol. 20, No. 6 (2010) 843–873
© World Scientific Publishing Company
DOI: 10.1142/S0218194010004967

World Scientific
www.worldscientific.com

EXPLOITING A GENERIC APPROACH TO CONSTRUCT COMPONENT-BASED SYSTEMS SOFTWARE IN LINUX ENVIRONMENTS

JÓ UEYAMA

Institute of Mathematics and Computer Science
University of São Paulo
São Carlos, SP, Brazil
joueyama@icmc.usp.br

EDMUNDO R. M. MADEIRA

Computing Institute, University of Campinas (UNICAMP)
Campinas, SP, Brazil

FRANCOIS TAIANI

Computing Department, Lancaster University
Lancaster LA1 4WA, UK

RAPHAEL Y. CAMARGO

Federal University of ABC (UFABC)
Santo André, SP, Brasil

PAUL GRACE and GEOFF COULSON

Computing Department, Lancaster University
Lancaster LA1 4WA, UK

Received 2 March 2009
Revised 9 September 2009
Accepted 26 October 2009

Component-based software engineering has recently emerged as a promising solution to the development of system-level software. Unfortunately, current approaches are limited to specific platforms and domains. This lack of generality is particularly problematic as it prevents knowledge sharing and generally drives development costs up. In the past, we have developed a generic approach to component-based software engineering for system-level software called OpenCom. In this paper, we present OpenComL an instantiation of OpenCom to Linux environments and show how it can be profiled to meet a range of system-level software in Linux environments. For this, we demonstrate its application to constructing a programmable router platform and a middleware for parallel environments.

Keywords: System software; component-based software development; middleware.

1. Introduction

Component-based technologies are nowadays widely used to develop application-level software, as illustrated by the numerous component-based platforms available to application developers (e.g. Mozilla plugins, Enterprise Java Beans, NET) [18, 20, 30]. Building on this success, a number of approaches has recently been proposed to apply component-based programming to system-level software. These approaches cover a wide range of systems, from embedded devices [8, 24] and operating systems [6, 10], through to programmable networks [2, 14, 19] and middleware platforms [5, 27].

This research has shown that components are well-suited for system software for three key reasons: first, they provide a high degree of modularization, thus allowing complex system software to be divided into simple and easy-to-develop functional parts. This "divide and conquer" approach facilitates the design, development and debugging of complex system software. Secondly, because of their inherent modularity, component-based platforms tend to be much more portable than monolithic systems. This is particularly advantageous for system software, whose portability is often inherently difficult to achieve and can provide a key competitive advantage in terms of software quality. Thirdly, components offer reusability, where new systems software can be constructed with less effort by assembling existing components.

Nevertheless, most of the efforts on component-based system software tend to be highly inflexible. More fundamentally, most of the component models are limited in terms of *target domain* (e.g. embedded systems, middlewares, operating systems or programmable networking environments). For example, OSKit [7], THINK [6], and MMLite [10] exclusively target component-based operating systems and cannot be used to realize programmable networking software or to implement a distributed middleware. Similarly, most approaches can only be deployed on conventional desktop machines, as opposed to less conventional environments such as PDAs and embedded devices. Those supporting embedded devices typically only support one type: Vera [14], NetBind [2] and NP-Click for instance are limited to the Intel IXP family of routers [12]. Although, these approaches are suitable for each target domain and deployment environment, we identify that they are unable to be applied to other system domains and environments.

Because of this narrow focus, existing component models fail to support systems that span multiple deployment environments, and cannot be used to develop platforms that combine multiple target domains. As a consequence, different environments and different target domains require different programming models and different APIs. This lack of generality inhibits skill transfer, prevents component reuse across platforms, and generally drives development costs up. To address the abovementioned shortcomings, we have proposed an approach called OpenCom [3, 33] which is a generic OS-independent component model that gives an infrastructure to construct systems to a wide range of domains which includes OSs,

middlewares and network stacks. In addition, OpenCom gives the support to develop system software aimed at a variety of deployment environments (PC, routers, sensor nodes).

This paper contributes with a reconfigurable generic component model for building systems software in Linux environments called OpenComL. In this paper, we outline the major features of this technology, present its implementation and show two case studies validating the generality and tailorability of our generic component-based systems building tool. OpenComL follows the OpenCom approach and constructs systems software at a minimal cost. This cost reduction is measured, for example, by the skill-sets transference across several development tools. A generic approach to develop systems targeted to a variety of domains substantially reduces the time to develop a system with a prior experience on a generic development tool.

OpenComL provides a minimal core on top of which the domain specific elements (e.g. Real-time-specific elements) can be added as component frameworks. This approach leads to more commonality and therefore more reusability. The design of our platform OpenComL was guided by three key features. *First*, OpenComL does not recommend any policy or facility that is specific to a particular target domain (e.g. real-time support or media-streaming support). When required, such features can be seamlessly added through a clear and principled extension mechanism.

Secondly, OpenComL's programming model can be easily ported to a wide range of deployment environments (as long as Linux is supported), from standard PCs, set-top boxes, and resource-poor PDAs, through to high speed network processor-based routers. This portability results from an incremental design based on a core set of minimal features. This minimality ensures this core can be accommodated even within the most constrained devices.

Finally, as our prototype evaluation shows, OpenComL incurs negligible performance overhead and has a very small memory footprint. This is particularly vital for deeply embedded software that runs on highly-constrained devices.

The remainder of this article is organized as follows: we first provide some background on component technology (Sec. 2) and review the related work along with the corresponding research challenges (Sec. 3). We then present the design choices and implementation of OpenComL's in detail (Sec. 4). In Sec. 5, we present a detailed evaluation of OpenComL based on the construction of two non-trivial systems: a programmable networking environment and a middleware for deadlock-free parallel applications running on Linux. Finally, we discuss our major contributions in Sec. 6 and conclude in Sec. 7.

2. Component Technology

This section introduces the main notions behind component-based software development. We start with basic concepts (components, interfaces, receptacles and

bindings), and then move on to component frameworks, which are largely employed by OpenComL.

2.1. *Components, interfaces, receptacles and bindings*

Most components models share the same basic common concepts [31], which are also used in OpenComL:

- *Components* are encapsulated units of deployment and functionality that interact with the outside world through interfaces and receptacles. A component may have multiple interfaces and receptacles.
- *Interfaces* are specifications of operations provided by a given component. Interfaces are defined by sets of operation signatures and associated datatypes. In OpenComL, they are expressed in OMG IDL [22] to support language independence.
- *Receptacles* are "required interfaces" and are used to make explicit the dependency of one component on other components. Interfaces and receptacles are collectively termed as *interaction points*.
- *Bindings* express an association between a single interface and a single receptacle. Different binding types can often be identified, depending on the semantic of the association, and the particular interaction protocol used between the two components.

2.2. *Component frameworks*

Component frameworks (hereafter CFs) have been defined as "*collections of rules and interfaces that govern the interaction of a set of components* '*plugged into*' *them*" [31]. A CF embodies rules and interfaces that make sense in a specific application domain. For example, a CF for multimedia protocols can consist of a variety of interfaces and rules to read various types of data (e.g. MPEG files, AVC files). Similarly, CFs can determine what constraints are required to govern the functionality of a set of components. For example, a packet routing CF can determine that a packet scheduler component must always read its input from a packet classifier. Such constraints are useful to ensure a meaningful reconfiguration which means that the system must provide support for describing them in detail.

Essentially, CFs provide the necessary support for the components they contain and regulate interactions between components according to the need of their specific domain of application. Because of their regulative nature, CFs are particularly well suited to dynamically adaptable systems. For example, one can implement a buffer management CF that accepts multiple buffer management policies at runtime. One can also implement a thread management CF that can plugin multiple thread management policies dynamically. A component framework can either operate alone or interact and cooperate with other CFs (as long as it conforms to the rules laid

down by the host CF). For this reason, in OpenComL CFs are themselves implemented as components.

3. Related Work and Research Challenges

We now briefly discuss several component models that we analyzed with a focus on the flexibility of the architecture in constructing systems independent of the target domain and deployment environment. The component models were classified into four distinct categories based on the domain of the target systems at which they were aimed: embedded systems, middleware platforms, operating systems and programmable networking environments. We use this overview to discuss the research challenges pertaining to the use of components in system software.

3.1. *Existing work*

In the first category a number of component models for embedded systems have been proposed (Koala [24], RoboCop [21], PECOS [37], and SaveCCM [8]). Nevertheless, none of them provides a generic programming model that is suitable outside of their targeted area. Most of them are primarily concerned with implementing a model that can ensure that the constraints for a real-time system are all fulfilled. For example, some of them (SaveCCM, PECOS, and RoboCop) provide a *component description language* that is particularly designed to ensure that all the informed constraints are fulfilled. However, this description language cannot be applied widely to other target domains since it is narrowly-targeted at real-time systems. Others such as the work in [26] applies component-based development to providing safe reconfigurations in the domain of embedded systems.

In the second category, component models for middleware platforms have been proposed (K-Component [5], DPRS [27], Fractal [1], and OMG's CORBA Component Model (CCM) [23]). These component models demonstrated that they have poor support for deploying multiple component styles, particularly components written in Assembly language. This precludes the adoption of these component models for primitive environments such as an embedded device. In addition to this, it is evident from this analysis that most of the above component models do not aim to deploy components in environments that are resource-constrained. This is normally reflected in the higher memory footprint overhead that is imposed by most of the analyzed component models.

In the third category, component models targeted at operating systems includes THINK [6], MMLite [10] and OSKit [7]. Most of the component models are primarily targeted at providing modular OS functionality and that this hinders the adoption of such technologies for constructing systems for other domains. Another factor is that some of the technologies investigated rely on heavy-weight platforms that preclude their use, particularly where memory resources are poor. For example, 2 K

is built on top of CORBA, whereas OSKit relies on a subset of Microsoft COM. This hinders the use of both 2 K and OSKit in resource-poor embedded devices.

In the fourth category, we also analyzed the component technologies targeted at programmable and active networks. This includes models such as NetBind [2], VERA [14], Shalaby's router [28], ACE [12], and Click [19]. This analysis suggested that these component models for networking environments are all narrowly-targeted at a particular router platform and unable to be employed to construct systems targeted at other domains. This reflects the goals and nature of these component models: they are usually targeted at solving one particular challenge in the domain of computer networks. For example, NetBind is a component model that is aimed at constructing data paths that are reconfigurable at runtime. ACE proposes a component model that is particularly targeted at deploying components on the Intel IXP routers.

3.2. *Research challenges*

A number of component technologies have been proposed in recent years as seen above. Although seminal, these efforts suffer from a number of deficiencies, which we discuss here in more details.

First, these technologies tend to be tightly coupled to *a particular target domain* (e.g. embedded systems, programmable networking environments, or middleware platforms). They are optimized for the construction of a particular family of systems and usually focus on one particular challenge: maximizing performance, minimizing memory requirements, allowing dynamic reconfigurations. NetBind [2], for instance, is limited to programmable networks development and focuses essentially on dynamic data-paths construction.

Similarly, these technologies tend to support *one particular deployment environment* only. The PECOS component model for instance [37] is limited to field devices; VERA [14] and NetBind [2] focus on network processor-based routers. More crucially, most of the remaining approaches have so far only been deployed in a conventional PC-based environment (e.g. K-Components [5], Click [19]). In addition, some of them (e.g. Koala [24], and VERA [14]) strongly depend on the underlying operating system and thus cannot be easily ported to other environments. It is important to stress that these efforts are suitable to the domain and environment that they were targeted to and that they provide all the benefits gained from the component approach. Nevertheless, most of these efforts are not straightforwardly applicable to a wide range of system domains and environments.

In terms of adaptation, many of these approaches do not support *run-time reconfiguration*. This is particularly inhibiting in embedded mobile devices where resources are particularly constrained and resources availability evolves dynamically. This is also a strong limitation for long-running systems where unplanned functionalities often need to be inserted without any interruption (e.g. the introduction of a new forwarder in a programmable router).

These technologies also have varying *memory footprint overhead* depending on the target system they were aimed at. This variability further restricts reuse and technology transfers. PC-based component models for instance tend to be rich in features (e.g. reflection, a large number of binding types). As a consequence, they have quite a high memory overhead and cannot be deployed in constrained environments [5]. Conversely, components models that target constrained devices tend to lack the rich features that are expected in unconstrained environments and cannot be reused in a PC-based context.

In this paper, we propose to demonstrate how these challenges can be addressed with OpenComL, a component model for Linux environments that follows the OpenCOM approach [3, 33]. OpenComL provides the necessary infrastructure to construct a wide range of systems for a variety of deployment environments. Thanks to its incremental design, OpenComL is inherently extensible to accommodate domain-specific facilities in a natural and explicitly-supported way. It also supports run-time reconfiguration, as illustrated with our case studies. Finally, because of its minimalist core, OpenComL incurs minimal overheads as shown in our performance evaluation.

4. OpenComL: A Generic Reconfigurable Component Model for Linux Environments

This section provides the key features of OpenComL, an instantiation of the OpenCom [3, 33] general-purpose component model for system software. OpenComL enables the construction of system-level software pertaining to a variety of domains (e.g. embedded devices, networking environments, middlewares) in Linux environments. We begin this section introducing OpenCom and examining its relationship with OpenComL and close this section with a discussion of the the potential benefits gained through this approach.

4.1. *OpenCom and OpenComL*

OpenCom [3, 33] is an OS independent component-based system building approach that has been used by several research projects. It is a generic component-approach that can be adopted to construct a range of system domains such as OSs, embedded systems and middlewares. Figure 1 shows the overall OpenCom architecture.

A key characteristic of OpenCom is that above the kernel all elements are optionally composed in the form of component frameworks (see Fig. 1). These elements are deployable by the kernel and extends the initial minimal architecture to the target OpenCom instantiation. For example, if one desires to instantiate OpenCom to sensor networking environments, the OpenCom platform should be designed without the extensions as sensor nodes are often resource constrained devices. On the other hand, if one wishes to instantiate OpenCom for Linux environments then, the platform itself should be designed with the extensions as they can serve as the means to tailor the systems to the target need and platform. In addition, Linux

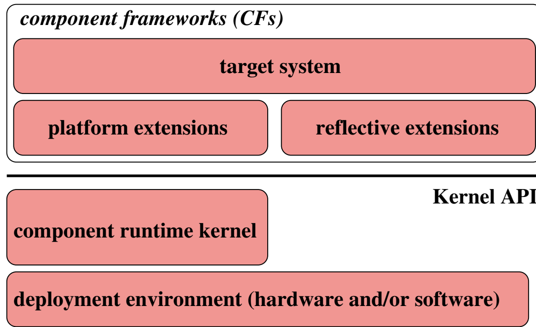

Fig. 1. Overall OpenCom architecture.

often runs in devices such as desktops and workstations where there are enough resources to support the extension components.

OpenComL is a component model particularly targeted to Linux environments that follows the OpenCom approach. It aims at validating the generality of OpenCom in constructing component-based system-level software for Linux in a range of devices such as PCs, PDAs programmable routers and embedded hardware. Since Linux distributions are available for a wide range of heterogeneous devices (e.g. from standards PCs to sensor nodes), we believe that it will be a good exercise to validate our generic approach.

For this, we provide a practical evaluation of OpenComL by showing that our approach can be successfully profiled to meet the needs of a range of devices running in Linux. We construct two non-trivial systems running in Linux environments that span two distinct domains. The first pertaining to the domain of programmable networks while the second to the domain of middlewares for parallel environments.

4.2. *OpenComL's architecture*

The OpenComL's architecture (Fig. 2) comprises three key elements: (i) a *runtime kernel*, (ii) an optional *extension framework*, and (iii) optional *reflective meta-models*. *Target system* represents the software developed by the programmer to execute over the OpenComL environment.

- The *component runtime kernel* is OpenComL's core element. It implements a minimal set of key operations to manage components and create bindings. Any additional functionality must be captured in the form of a component framework (CF) and implemented as an external component that is deployed at runtime on top of the kernel. This applies in particular to the two other key elements of OpenComL, the reflective meta-models and the extensions framework.
- The *extensions framework* is a component framework (and hence also a component) that provides structured support for extensibility within the deployment environment. This framework is required whenever heterogeneous types of

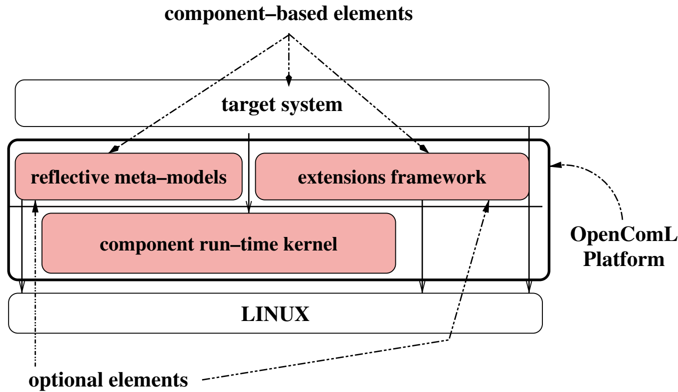

Fig. 2. OpenComL Core Elements.

components are simultaneously needed. Indeed, the kernel itself is only capable of deploying one type of components termed *primary-style* components. The particular nature of *primary-style* components depends on the implementation of the kernel.

- The *reflective meta-models* layer consists of components that facilitate system reconfiguration by allowing different aspects of the system to be inspected, adapted and extended at runtime. For example, an architecture meta-model exposes the system topology in terms of the deployed components and the connections established between them. This topological representation can be inspected at the system reconfiguration.

Figure 2 shows the OpenComL's architecture. The layers above the *component run-time kernel* (i.e. the reflective meta-models, the extensions framework and the target system layers) should be implemented as OpenComL components and are therefore deployable by the kernel. The arrows between the layers indicate the allowed interactions (e.g. *target systems* are allowed to invoke functions implemented in the *component run-time kernel* layer).

4.3. *The OpenComL kernel*

The OpenCom kernel follows a *microkernel-style* design. This implements the minimal required functionality of the Kernel API (see the supported operations in OMG IDL in Fig. 3) which in summary supports the life-cycle of components, i.e. the API gives support to load, unload, bind and unbind components. Significantly, the kernel is restricted to deploying a particular style of component called the *primary-style* component, which in OpenComL were chosen to be C++ XPCOM components [20]. Limiting kernel functionalities ensure actual implementation will only require minimal amount of code, thus making them easier to change and port it to other environments.

```
1  typedef short status;
2  typedef long GlobalID;
3  interface IKernel {
4     status load(in GUID componentType, out GlobalID ID);
5     status unload(in GlobalID ID);
6     status instantiate(in GlobalID ID, out GlobalID compID)
          ;
7      status destroy(in GlobalID compOrBindID);
8      status bind(in GlobalID interface_ipnt, in GlobalID
          receptacle_ipnt, out GlobalID bindingID);
9      status putprop(in GlobalID entity, in string key, in
          any value);
10     status getprop(in GlobalID entity, in string key, out
          any value);
11     status notify(in GlobalID compID);
12   };
```

Fig. 3. The OpenComL Kernel API.

An OpenComL kernel is limited to a single address space, but permits the usage of multiple address spaces through the *extensions framework* (Sec. 4.4). This approach makes OpenComL kernels lightweight and deployable in resource-constrained devices (e.g. a sensor node) where support for multiple address space is often not required.

The kernel is made up of the kernel *registry* and the implementation of the API operations. The registry maintains a minimal meta-data representation of the currently deployed components, interfaces, receptacles and bindings. In terms of the API, the key distinction between *Load*() (line 4 in Fig. 3) and *Instantiate*() (line 6) operations is that the former reads component meta-data from the library in which the requested component is packaged, while the latter utilizes this meta-data to create an instance of this component.

The *putprop*() and *getprop*() (lines 9 and 10) operations provide the necessary support to gain access to the above mentioned kernel registry. In both operations, the *entity* argument is the unique identifier of a component, interface, receptacle or binding. The *key* argument is the property name arbitrarily assigned by the programmer, while the *value* corresponds to the value of the property. An Interface is bound to a receptacle using the *bind*() operation (line 8). This operation enables the construction of systems by selecting components and connecting them up according to the needs of the target system. The disconnection between components is effected through the *destroy*() operation (line 7).

The OpenComL kernel also supports reconfiguration at runtime, through a minimal approach to reflection [17]. The *notify*() operation (line 11) registers a component that should be *notified* whenever an operation in the Kernel API is invoked. For example, if *notify*() registers a meta-architecture component then each Kernel API call (e.g. *load*(), *bind*()) will be *notified* to the registered meta-architecture.

OpenComL is implemented in C++, with the primary-style components implemented as Linux *shared object* libraries [16], allowing them to be deployed dynamically at runtime. This leads to the provision of a dynamic re-configurable component model in which required policies are incorporated into the system dynamically.

4.4. *The extensions framework*

The role of the extensions framework is to provide structured support for extensibility at the deployment environment level. This framework is required whenever special implementation of components needs to be deployed. As mentioned in Sec. 4.3, OpenComL is only capable of deploying primary-style components implemented in C++ and packaged as Linux shared libraries and therefore an extension is required to make other style of components deployable.

The extensions framework is a component framework that accepts three types of plugins: *caplets*, *loaders* and *binders*. These components are OpenComL primary-style components that embody all the necessary functionality to deploy components in a particular environment (e.g. a loader can be designed to load and instantiate components written in Assembly language for an embedded device). The resulting 'domain of deployment' that is supported by a kernel as a result of these plugins is what we call a *capsule*.

- *Capsules* are "component containers" that may encompass multiple address spaces. They provide a namespace in which all components and interfaces can be individually identified by a unique identifier. Each capsule has one single instance of the OpenComL kernel that is responsible for deploying components in that particular capsule. Figure 4 shows a capsule that embodies four components, binded through *receptacles* (the small cups) and their corresponding *interfaces* (the small circles).

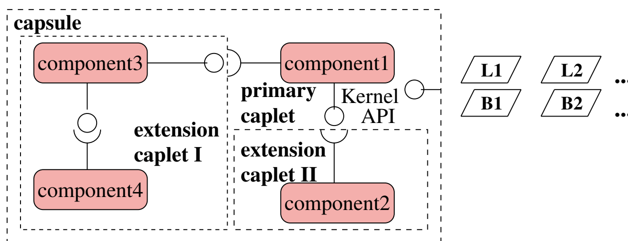

Fig. 4. The elements of the OpenComL programming model.

- *Caplets* are OpenComL components that provide an isolation mechanism between components which are mutually distrustful or which have different privileges. Caplets are also designed to support different technology domains in the underlying deploying environment. For example, if a system requires deployment of both C++ and Java components, this can be achieved by creating separate caplets for each technology domain. Figure 4 shows several caplets in which components are loaded in. Each caplet isolates the components loaded in that caplet from components loaded in other caplets.
- *Loaders* provide multiple-loading mechanisms, allowing a wide range of component styles be deployed in a heterogeneous environment, such as an embedded device. These loaders, represented in Fig. 4) as parallelograms labelled "L", encapsulate the complexity in loading components in such an environment.
- Finally, *binders* are OpenComL components designed to allow developers to implement any "binding mechanism" that might be required in the underlying deployment environment. For example, one can implement a binder that creates connections between Java components and another that connect components written in Assembly language. Binders are shown in Fig. 4 as parallelograms labelled 'B'.

Initially, there only exists a single caplet which coincides with the enclosing capsule and is called the *primary caplet*. Additional caplets, called *extension caplets*, may be progressively instantiated to support different component styles and thus extend the scope of the capsule. By default, the kernel-level API is only capable of loading and creating bindings between primary style components within the primary caplet. This loader and binder are known as the *primary loader* and *primary binder*, respectively. Other implementations of loaders and binders may be instantiated to load and bind components of other style of components residing within the same caplet or across different caplets. These loaders and binders are called *extension binders* and *extensions loaders*, respectively.

Figure 5 shows the deployment and connection between a primary-style component (PrimaryComp1) and a Java-based component (JavaComp1). The primary-style component is deployed by the OpenComL kernel and the Java-based one by the *MyJavaLoaderID* loader. The binding between the deployed components, which run in different address spaces, is carried out by the *MyJavaToPrimaryBinder* binder. The binding is conducted through a communication channel, implemented using Linux pipe channels, that enables the communication between the two caplets. The *MyJavaCaplet* caplet is created by *MyJavaCapletCreator* component that *forks* a new process for the Java caplet.

The example above illustrates the deployment of primary-style and Java-based components, but OpenComL is capable of deploying components of any style including the ones written in Java, C/C++, and Assembly language. This is possible owing to loaders and binders component plugins. This feature is particularly interesting for embedded devices that have a heterogeneous environment, such as an NP-based

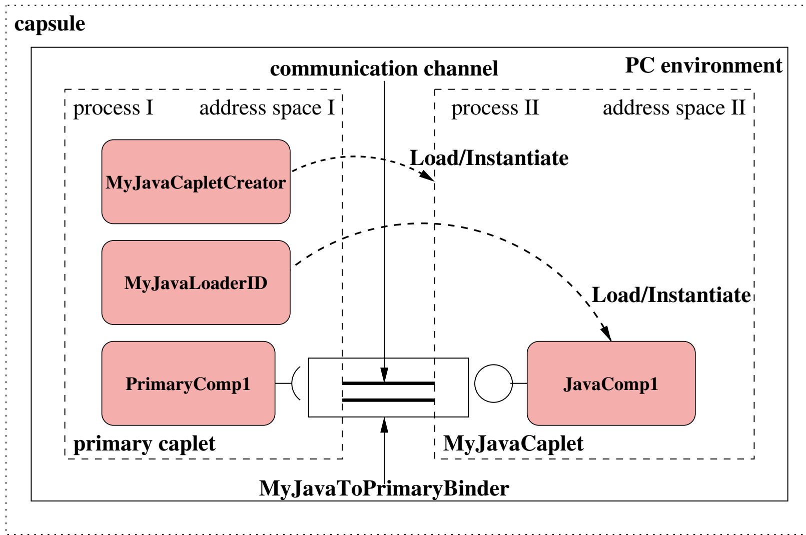

Fig. 5. An example of caplet, loader and binder in Linux environments.

router (see the study case in Sec. 5.1). In such a heterogeneous environment, loaders and binders can be implemented to deploy components for each environment. Finally, since OpenComL kernel supports dynamic configuration, caplets, loaders and binders are deployed at runtime and unloaded when no longer required.

4.5. *Reflective meta-models*

Reflective meta-models implement the necessary functionality to inspect, adapt and extend the target system at runtime. In addition, since the meta-models are themselves components, their precise configuration can be tailored to the needs of the target system. As the reflective meta-models are merely primary-style components they can be loaded and unloaded by the kernel in the same manner that the target systems are deployed. It is important to mention that the term meta-models is derived from the notion of "meta-space" in reflective systems [29] and is unconnected with the term "meta-model" from mainstream software engineering practice.

OpenComL includes two meta-models: architecture and interface meta-models. These two orthogonal meta-models are optional and can be dynamically loaded whenever demanded and destroyed when no longer required. As they are deployable on-demand, it helps us in reducing the resource consumption of OpenComL.

Other meta-models can also co-exist besides the two just-mentioned. For example, the interception meta-model which is proposed in OpenCom [3] can provide a principled way to insert arbitrary code (called interceptors) within bindings. Interceptors are executed either before or after the invocations takes place across the bindings.

4.5.1. *Architecture meta-model*

The *architecture meta-model* uses structural reflection to represent the current topology of the system. In OpenComL, the architecture meta-model consists of a representation of components and their bindings. The main value of this representation is that it allows the topological adaptation of a target system. For example, in a media stream application scenario, the meta-model could be used to replace an MPEG codec component with an H.26 codec in a dynamic way, when a mobile PDA migrates from a fixed to a wireless network. This can be achieved by first, inspecting the underlying system topology to locate the codec component, and then adapting or extending the topology to effect the corresponding change at the base-level (i.e. to replace the codec component). The implemented meta-model builds a graph in which components are represented as nodes and bindings between them as arcs. The meta-model inspects the architecture by transversing the graph; while the adaptation is carried out by adding or removing the node and or an arc. For example, an arc is removed between nodes whenever a binding between components is destroyed.

We implemented the meta-architecture as a primary style component. This component maintains a graph representing the system topology, which adopts the same data structure implemented in the registry, i.e. an ID followed by a name and value pair. The implementation of the architecture meta-model depends on the kernel's *notify*() call, which is invoked to register the meta-architecture component. Once the registration has been made, every kernel call (i.e. *load*(), *bind*(), etc.) causes the invocation of the callback functions that are implemented on the registered meta-architecture, which in turn update the system topology for maintaining a complete architectural representation of the capsule.

4.5.2. *Interface meta-model*

The *interface meta-model* supports two different capabilities: (1) allows discovering details of the interfaces and receptacles in terms of operation signatures at runtime and (2) enables "dynamic invocations" to be undertaken on dynamically-discovered interfaces. These capabilities enable components to invoke interface types which are unknown when they are coded. The interface meta-models are particularly beneficial in target systems that need to support functionalities such as debugging, and component database browsing.

In terms of the implementation, the interface meta-model is a primary-style component that is loaded and unloaded in the same manner as components of the target systems.

The meta-model uses the *getprop*() and *putprop*() calls implemented in the Kernel API to store and retrieve information of interfaces and receptacles associated with each component. For example, the segment of the pseudo-code in Fig. 6 allows retrieving the name of an interface whose ID value is 1 to the variable *interfaceName*.

```
1   typedef long GlobalID;
2   GlobalID intID = 1; /* This is the interface identifier
      */
3   char * interfaceName = NULL;
4   getprop(intID,     /* interface ID */
5           ``name'', /* name of the attribute */
6           &interfaceName) /* content of the `name'
                attribute */
```

Fig. 6. An example of use: *getprop*() operation.

The *getprop*() and *putprop*() calls also allow implementing dynamic invocations, where interfaces are retrieved at runtime.

Interfaces are retrieved dynamically through the *QueryInterface*() operation, which is implemented in the *ocISupports* interface. *QueryInterface*() discovers what interfaces are implemented by a given component and allows switching from one interface to another and back again, if required. This call receives as parameters a reference to a universal unique ID, which names a capsule-scoped entity, and an output parameter employed to hold a pointer to the requested interface.

4.6. *Benefits of our approach*

We now underline a number of potential benefits when adopting a generic component-based approach in Linux environments.

- *Skill transference.* The utilization of different technologies to build applications for each target device and applicability does not allow transference of skills across different tools. Skill sets and areas of expertise are rarely transferable when relying on different technologies. A generic approach promotes skill transference as developers only utilize a single tool for developing applications based on a variety of technologies.
- *Costs.* The employment of a unique tool can also lower the software costs that are required when purchasing a variety of technologies for a wide range of mobile environments. This is particularly beneficial in heterogeneous environments such as a network-processor-based router.
- *Minimizing re-invention of the wheel.* The lack of generality often leads to reinvention of the wheel. OpenComL is sufficiently general to ensure that there is no need for behaviour (such as dynamic adaptation) to be repeated in an ad-hoc manner across technologies.
- *Addressing the failure to construct generic system software.* Narrowly-targeted tools fail to construct applications that spans software technologies and hardware platforms.

- *Code Reusability/modularity*. The generic approach fosters code reusability as developers may reuse existing components and/or processes. For example, in our approach the notion of loader and binder let us reuse existing Microcode components (Microcode is the Assembly language for IXP1200 family routers), JCSP and occam-pi processes.

5. Case Studies

In this section we present two case studies whose overall goal is to evaluate the flexibility of our component model in supporting the development of systems software in a wide range of environments and domains. The case studies illustrate the capacity of the extension components (i.e. caplets, loaders and binders) to deploy a variety of component styles in diverse environments. This demonstrates that OpenComL is suitable for constructing systems that run in demanding heterogeneous environments such as a network processor based router.

We also evaluate the extent to which OpenComL can run successfully in resource-constrained environments. Because OpenComL is based on Java and Linux, it clearly cannot fit onto extremely resource-scarce platforms such as sensor motes [32]. However, our first case study demonstrate that it runs well in a moderately resource-scarce environment, the IXP1200 Network Processor that we describe below.

5.1. *Case study I: Programmable networking environment on network processors*

Network processors (hereafter called NPs) consist of a multi-processor programmable device that forwards and processes packets at high speed (typically gigabit/s). Like conventional CPUs, these processors are programmable. However, unlike conventional CPUs, NPs are optimized for packet forwarding and processing. They offer an ideal case study to validate the proposed generic systems-building technology for the following reasons:

- NPs are widely known to be very difficult to program. For example, they have their own hardware architecture, often with no OS and rely on a specialized language to program these devices. They also operate in a heterogeneous environment with multiple types of processor and memory.
- The developed system must be extremely efficient to meet the requirements set by these devices, namely to meet the imposed gigabit forwarding speeds.

The goal of this case study is to demonstrate that (1) the OpenComL programming model can be used to construct systems in NPs, such as the Intel IXP1200 [12] and (2) that the overhead imposed by OpenComL can be regarded as negligible.

Intel IXP1200 is composed of: (i) a StrongARM CPU running Linux, which acts as the general control processor in the router; (ii) an array of six so-called

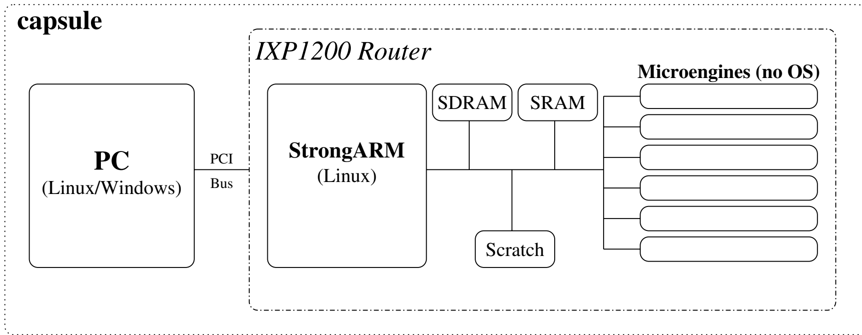

Fig. 7. Intel IXP1200 network processor.

microengines RISC CPUs that are attached to each other and share three types of memory as shown in Fig. 7 (i.e. Scratchpad, SDRAM and SRAM); and (iii) a set of dedicated hardware elements (not shown in the figure) such as the network interface and hash unit.

We mapped OpenComL onto the IXP1200 as follows: we hosted a primary *caplet* in the StrongARM environment as it is directly connected to the microengines and the PC, as well as being more central from the point of view of the router. The caplets are realised using Linux OS processes. We devised two loaders for the IXP1200 environment: the *primary* loader and the microengine loader. The former deploys primary-style components and the latter Intel's Microcode components. Similarly, we developed two binder components: the *primary* binder (for connecting primary style components) and the microengine binder. In the following we discuss the microengine implementation of our OpenComL prototype in more detail as they concentrate most of complexity of this environment.

In more detail, a *microengine caplet* provides a communication channel between the primary caplet in the StrongARM and the target microengine caplet. This channel is implemented through the libraries provided by Intel that support direct access from the StrongARM to the microengine microstore, and to memories such as the Scratchpads, SDRAMs and SRAMs. Instructions and data can be exchanged between components in the StrongARM and components in the microengines by sharing these registers.

A *microengine loader* embodies the capability to load and instantiate microcode components. The components that are deployable by OpenComL in the microengines are coded either in microengine-C (i.e. a subset of C for the IXP platforms) or Assembly. This loader enables the target developer to deploy components in the same fashion that takes place for the primary style components, despite the primitive underlying environment.

Finally, the *microengine binder* connects microcode components that are running in the same microengine. This is effected by "morphing" a jump instruction where one component directs its execution to the entry-point of the next component. This

approach was pioneered by the NetBind project [2]. The required entry and exit points are all stored in the kernel registry and handled through the kernel *putprop*() and *getprop*() calls.

5.1.1. *Performance and overhead*

This subsection examines a quantitative evaluation of OpenComL for this case study which includes the overhead incurred by OpenComL. Most measurements were obtained by averaging a million experiments, such as when measuring the loading time of OpenComL primary-style components through the *load*() call.

The experiments in this section were performed using a workstation with four 1.6 GHz Pentium CPUs, 512 MB of RAM, and running Linux 2.4. Some experiments were also carried out on the *Radsys* IXP1200 router environment [12], a network processor-based router that runs *BlueCat* embedded Linux as its OS. In all experiments, OpenComL and the underlying OS are the only software that executes on the workstation and the IXP1200 router environment.

The OpenComL kernel for this evaluation was implemented in C++ and compiled using the GNU GCC compiler. Primary-style components deployed by this kernel are stored in Linux *shared object* libraries, which enable components to be deployed dynamically at runtime. The kernel uses the C++ object creation mechanism for component creation. On top of the kernel, optional extensions framework and reflective meta-models are also implemented in C++.

Memory footprint

The measurements for the minimal memory footprint overhead required to deploy the kernel itself was 32 Kbytes. This is a modest overhead and let us deploy it in a wide range of environments, including those where resources are constrained, such as a sensor networking environment. The kernel includes all the operations of the kernel API (e.g. *load*(), *instantiate*(), *bind*()) and the OSPL (operating system portable layer), a layer that maps each OS-level call (e.g. memory allocation functions — *malloc*() call) to the corresponding one in the underlying operating system.

Figure 8 shows the memory footprint imposed by the runtime of OpenComL and compare it with other component models. NetBind has been included in this comparison because its *code morphing* binding abstraction has been adopted in the implementation of an extension for binding microcode components. MMlite is a well-known component model for OSs (therefore a good point of comparison) and has a minimal runtime that is deployable in resource-constrained devices. SaveCCM is also a component model targeted at resource-scarce embedded devices that have a low memory overhead. The approach adopted by THINK regarding its flexibility is similar to that adopted by OpenComL. Finally, we show the memory overhead of the kernel from Open**COM** middleware, which is based on Mozilla's XPCOM.

Figure 8 illustrates that OpenComL, THINK and MMLite are deployable in a wide range of deployment environments since they both only incur a minimal

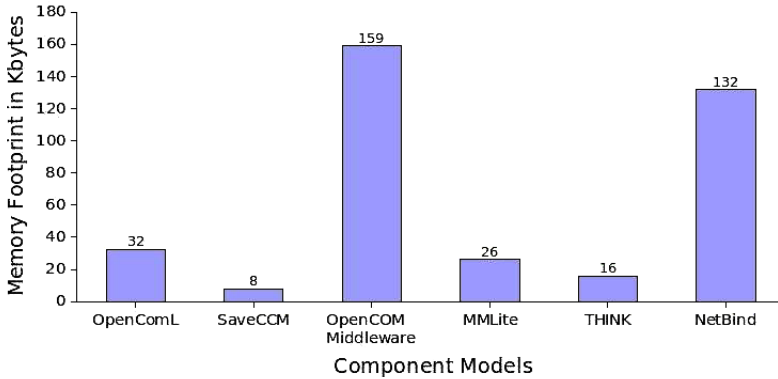

Fig. 8. Kernel's memory footprint comparison.

memory overhead. THINK's kernel runtime has only 16 Kbytes [9], but its kernel is particularly targeted to OS domains. SaveCCM provides the lowest memory overhead, but does not support runtime reconfiguration. OpenCOM and NetBind incur a high memory overhead which may hinder their deployment in resource-scarce environments. The high memory overhead of OpenCOM derives from the XPCOM dependencies and NetBind has an interpreter that is employed to decode all the deployment instructions (i.e. load, bind, unbind, etc.).

We have also computed the memory overhead required by a single component in OpenComL. The memory footprint to accommodate a single null primary-style component, with no interface, no receptacle and null initialization and finalization routines, was measured as 24 bytes. This result is comparable to the 20 bytes required for a single null C++ object. The per-interface and per-receptacle memory overhead is 8 bytes, with additional 5 bytes per operation.

Component loading and instantiation time

The time measured to load a single null primary-style component is 9.8 μs. This includes the time taken to read a shared object library from the disk storage and extract the information of each primary-style component stored in the library. The extracted information includes the constructor function, which is utilized to instantiate a component. This measurement was computed by calculating the average figure from over a million *load*() calls. This is comparable to that obtained when loading a null C++ object by means of the same shared object library, which was measured as 7 μs.

To instantiate a null already-loaded component, OpenComL took 0.47 μs compared to 0.28 μs taken to instantiate a null C++ object. The differences in this measurement can be attributed to the larger file size that is required by OpenComL components to accommodate information for the meta-data. In addition, it can also be attributed to the complex instantiation mechanism that is employed in

OpenComL, which includes the interaction with the kernel registry (see Sec. 4.3 for kernel registry).

Performance of the extensions framework mechanisms

Performance evaluation of the extensions framework mechanisms was carried out in the IXP1200 router environment.

We have developed a caplet, loader and binder components to deploy Microcode components that run in the microengines. Microengines are RISC CPUs that are targeted at packet forwarding and processing, while Microcode is the Assembly language for the microengines.

To evaluate the performance of the implemented microcode loader and binder plugins, a comparison was made with that obtained by NetBind [2]. NetBind is a good point of comparison given that our microcode loader and binder employ an approach that was pioneered by their project. NetBind loads at the granularity of so-called *pipelines*. A pipeline is a pre-configured assembly of singleton microcode components. These pipelines constrain the modules to be deployed in a linear topology and are created whenever these modules have to be loaded and bound. Unlike NetBind, our implemented loader and binder extensions allow the components to be deployed individually by creating any kind of arbitrary topology.

This experiment evaluates the imposed overhead to incrementally extend the linear topology of components in a single microengine. In particular, the time taken to include each microcode component in the existing pipeline is measured. The results given in Fig. 9 show that NetBind incurs an increasing linear overhead. This results from having to rebuild the entire pipeline for each new configuration. Unlike Netbind, the implemented binder plugin incurs a constant overhead for each extension, as the components are bound incrementally. The binding overhead is minimal as it only involves changes in the branch instructions. It should be underlined that in this experiment, all microengine components were deployed in a single microengine.

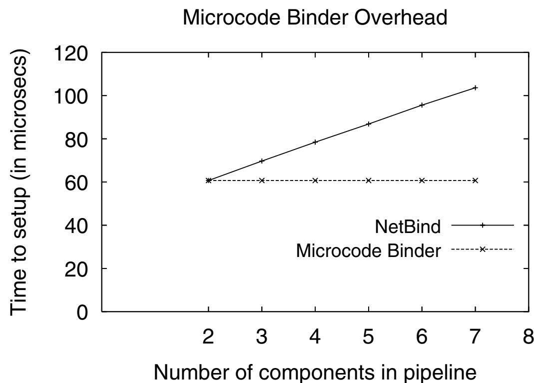

Fig. 9. Time to make an incremental addition of a component to an existing configuration.

5.1.2. *Overall evaluation*

This case study evaluates the reconfiguration facility and the ability to handle a heterogeneous environment, both of which are supported by OpenComL. This has been achieved because the programming model and the extension components allow multiple component styles to be deployed at runtime.

In addition, this case study validates the use of a component-based programming model for constructing system software in complex and primitive deployment environments, such as an NP-based router. In particular, the proposed programming model saves the developer from having to handle a wide range of deployment environments, which is commonly found in a programmable networking environment. Instead, the developer only sees components, loaders, binders and caplets that are operated in a standard manner. This uniformity, in which components are loaded, instantiated and bound in the PC, StrongARM and microengines through a unique programming model, eases the problem of programming these complex and heterogeneous environments.

5.2. *Case study II: Middleware for parallel environments*

This case study applies the OpenComL programming model to the domain of a middleware platform for parallel environments called FlexPar [34]. Essentially, it aims at evaluating the use of OpenComL in constructing non-trivial systems such as a flexible middleware that is capable of building parallel software at runtime, indicating that OpenComL is a generic form of technology that can be adopted to construct systems for a wide range of target domains.

FlexPar is a reconfigurable middleware that is capable of constructing parallel software at runtime. It creates applications by deploying multithreaded components that make use of the CSP (Communicating Sequential Processes) [11] disciplines. CSP is a paradigm for concurrent programming that prevents common problems found in concurrency, such as deadlocks, and race hazards. In the current prototype, FlexPar deploys components in JCSP [25] and occam-pi [35] as both support the CSP paradigm. JCSP is a set of Java libraries that provides CSP disciplines for Java programmers. occam-pi was devised at the University of Kent and is essentially an extension of the occam language [15], which also implements the CSP platform. It produces less overhead (e.g. in terms of the memory footprint) than Java and thus, occam-pi is often adopted to construct parallel embedded systems in environments with low resources [13]. On account of the programming language diversity and also in view of complex and unpopular languages such as occam-pi, we propose the use of OpenComL to construct FlexPar, which has been implemented in the following way:

- *Kernel*. In this case study, we coded the kernel in Java in order to simplify the deployment of components written in JCSP. The construction of the kernel proved to be straightforward, on account of its size and simplicity. The FlexPar kernel

(in Java) required a minimal memory footprint of 60 KBytes, which ensures that it is deployable on moderately resource constrained devices.

- *Caplets, loaders and binders*. Caplets, loaders and binders were implemented to support a range of component styles, including the primary (in C++) and interpreted (in Java and JCSP) styles. Thus, extensions (i.e. caplets, loaders and binders) to support primary-style components and Java-based components were employed in this case study.

The FlexPar architecture is implemented using OpenComL and is structured as depicted in Fig. 10. The architecture is composed of a microkernel whose role is to deploy the JCSP and occam-pi loaders and binders. The microkernel style to enable its deployment in environments where resources are particularly scarce.

There is no bias for JCSP and occam-pi in FlexPar, since the architecture is extensible and supports other parallel languages. This is illustrated in Fig. 10 through *x loader* and *binder* that are designed to load and bind, respectively components written in a fictitious *x* language.

Another key advantage is that FlexPar provides a unique tool for constructing parallel software relying on a variety of platforms, as shown in Fig. 11. A unique tool for constructing applications in JCSP and occam-pi (or any other language platform)

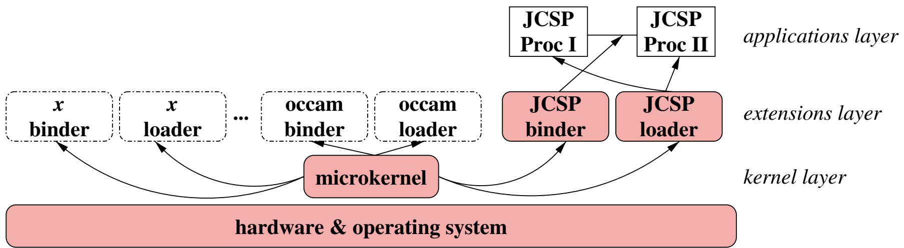

Fig. 10. FlexPar architecture.

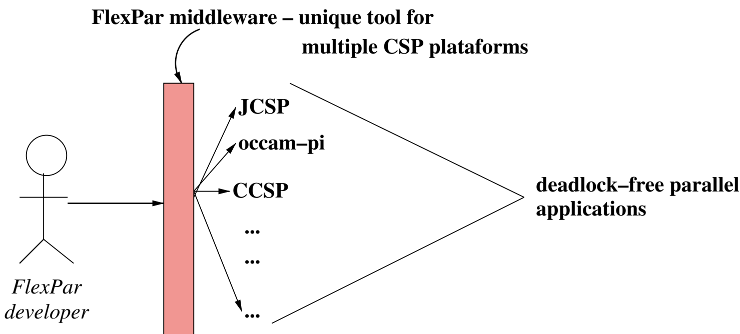

Fig. 11. FlexPar—unique tool for constructing parallel applications based on a variety of technologies.

can simplify the task of developers once they do not need acquire knowledge of multiple tools. FlexPar abstracts all the complexity of the underlying environment through loader and binder extensions that are deployable at runtime. Developers make use of these extensions to create applications in a variety of CSP platforms. The problem in relying on multiple tools is that skill sets and area of expertise are often not transferable across different tools.

Sensor networks is an application scenario where FlexPar can be adopted as a unique tool to overcome the heterogeneity of deployment environments. Sensor networks often comprise a heterogeneous environment with sensors from a variety of manufacturers (e.g. Gumstixs, MOTES Sky, etc.). The OpenComL component model, adopted by FlexPar, can help developers to deal with such a heterogeneity by providing loaders and binders that are targeted at each kind of sensor architecture. Another application is programming LEGO Mindstorms, where the flexibility of Flexpar allows different functionalities to be deployed and removed dynamically, such as selecting between vision or sonar navigation. Both scenarios (sensor networks and LEGO Mindstorms) require running parallel applications in relatively resource-scarce environments.

5.2.1. *JCSP loader and binder*

While the JCSP loader instantiates JCSP processes, the binder forms connections between processes in JCSP. These extensions were straightforward to implement as JCSP is already in Java. As the kernel is also in Java, we extracted the loading mechanism employed by JCSP and encapsulated the loading functionality in a separate component. This component, which embodies the loading functionality, can then be deployed in a dynamic way to load the JCSP processes.

The binder extension creates a binding between the JCSP processes. That is, it creates a communication channel between the two processes. Like the JCSP loaders, the binder is a component that can be deployed by the kernel. As JCSP includes several channel types (e.g. one to one channel, one to many, many to many), it is possible to have multiple JCSP binder implementations so that the FlexPar users can choose a suitable one for the target application. In brief, a *one to one channel* is a type of channel that connects one JCSP process to another, while the *one to many* style connects one process to multiple processes.

5.2.2. *occam-pi extensions*

occam-pi extensions include a set of components that enables FlexPar developers to create occam-pi parallel applications at runtime. While existing occam-pi development tools (e.g. KRoC [36], SPoC [4], Transterpreter [13]) give little support for dynamic reconfiguration, FlexPar allows one to create parallel applications in occam-pi dynamically at runtime. This is possible owing to the loader and binder plugins which essentially embody an occam-pi interpreter (based on the

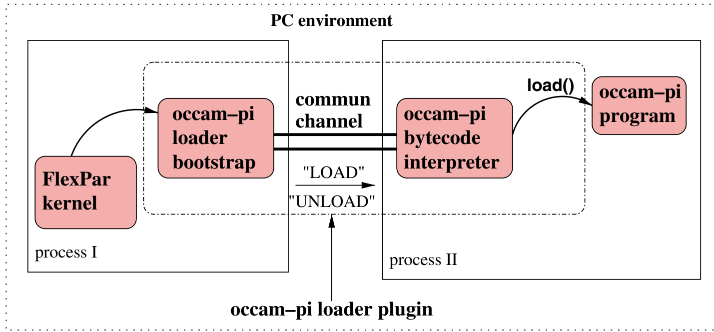

Fig. 12. Implementation of the occam-pi loader extensions.

Transterpreter, which is a virtual machine that executes occam-pi bytecodes) that parses and executes each bytecode instruction at runtime. The bytecode is generated by SKRoC that is part of the Transterpreter toolset.

The bytecode interpreter runs in a separate thread of control that is spawned by the extension itself. The configuration of a given application (e.g. which occam-pi code to load, which channel to connect to) is formalized by a series of keywords that the loader is capable of understanding. For example, *loader.sendCommand* (“*LOAD*”+ *codeName*) requests the loader to load the occam-pi bytecode called *codeName*.

Figure 12 shows that the FlexPar kernel also instantiates the occam-pi loader bootstrap which initializes all the elements that comprise the loader. Prior to running the interpreter, a communication channel is established between the loader and the interpreter.

Similarly, the occam-pi binder also forks a new thread that in turn listens to the binding instructions. For example, *binder.sendCommand*(“*BIND*” + *channelNameFrom* + *channelNameTo*) requests the binder to make a connection between the *channelNameFrom* and *channelNameTo* channels.

5.2.3. *Performance and overhead*

The experiments outlined in this section were all carried out in a Dell OPTIPLEX GX 620 series workstation with 3.6 GHz Intel P4 660 CPUs and 1 GB of RAM. With regard to software, all the measurements were collected by means of FlexPar implemented in Java running on Linux Fedora 5.0. These experiments consider that the workstation is only running FlexPar on top of the underlying operating system.

Memory footprint

We evaluated the memory footprint to ensure that FlexPar is well suited to relatively resource-constrained devices. As mentioned earlier, the kernel in Java required

Table 1. Memory footprint figures of the kernel & extensions.

Kernel & extensions	Memory footprint
Kernel	60 KB
occam-pi extensions	30 KB
JCSP extensions	8 KB

around 60 Kbytes to be deployed. Table 1 provides a summary of the memory overhead required by our prototype implementation.

The occam-pi extension memory overhead includes the loader and binder bootstrap along with the bytecode interpreter that translates and executes each bytecode instruction. The bootstrap creates a communication channel and initializes a new process along with an instance of the interpreter. It should be noted that in this implementation, a single bootstrap initializes both the occam-pi loader and the binder extensions. The JCSP extensions are almost negligible as most of JCSP complexity is encapsulated within each process. Overall, the figures above seem feasible for reasonably resource-constrained devices as FlexPar appears to be deployable in a wide range of these environments.

Component loading and instantiation time

This section discusses the overhead incurred by FlexPar's microkernel to instantiate and create connections between the extension plugins. For this particular evaluation, we deployed the JCSP loader extension to measure the overhead incurred by each individual operation found in Table 2.

Table 2 shows that *createInstance*() operation requires the highest overhead. This is primarily caused by the access provided to read the loader or binder from the disk storage. However, this overhead can be considered to be insignificant because creating an instance of the loader only takes place occasionally when the FlexPar architecture is extended. Thus, its impact on the overall performance can be regarded as negligible.

Table 2. Performance overhead to deploy the extensions plugins.

Operations	Time in milliseconds
createInstance()	30
destroyInstance()	3
bind()	4
unbind()	2.5

Commstime benchmark

Commstime is a timing benchmark that was devised at the University of Kent and is part of the JCSP library [25]. It consists of four parallel processes that send out 0

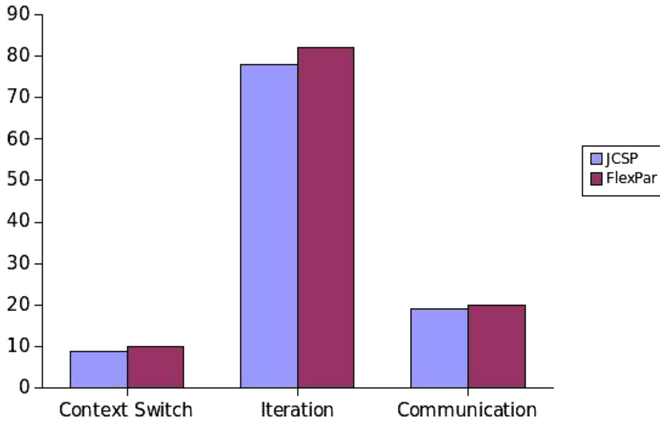

Fig. 13. Overhead incurred by JCSP and FlexPar.

and forwards whatever it receives through the channels. By measuring the elapsed time, it can compute the context switch overhead between processes, an average interaction time, and the communication overhead. Since there is essentially no work involved in this system other than message passing, it can be adopted to measure the context switch and communication overhead. The iteration time is an average among eight channel reads and writes.

Figure 13 shows the results obtained by the Commstime benchmark. They were achieved using the JCSP loader and binder to deploy JCSP processes. To provide an accurate result, we measured the average of several computations. The graph in Fig. 13 illustrates the comparison between JCSP and FlexPar. It shows the context switch, iteration and communication times incurred when JCSP processes are deployed by the JCSP platform and the FlexPar middleware. The measurements demonstrate that FlexPar incurred a negligible overhead (around 5% slower) when compared to that of the JCSP platform. This overhead can be primarily attributed to the FlexPar microkernel and the JCSP loader and binder that were required to run this experiment. Despite this negligible overhead, FlexPar (unlike JCSP) provides a flexible platform and allows developers to rely on multiple tools to create parallel applications.

5.2.4. *Overall evaluation*

The goal of this case study was to show that OpenComL could be used to better support the often challenging development of parallel software for a wide range of environments and applications, and with minimum resource overhead.

The lightweight kernel allows us to deploy the FlexPar middleware on devices where resources are reasonably scarce. It also helps us to port FlexPar to other environments. Owing to the notion of loaders and binders, FlexPar developers are able to deploy parallel processes coded in a variety of programming languages.

These extensions enabled us to rely on a single tool to deploy heterogeneous styles of processes. Loader and binder extensions encapsulate the whole complexity of the underlying target deployment environment at which they are aimed. This simplifies the development of parallel applications since FlexPar developers only see loaders and binders and they do not need to be aware of the full complexity of the deployment environment. For example, if one wishes to deploy occam-pi processes in the LEGO, an occam-pi loader and binder can be used for that particular LEGO to deploy these kinds of processes.

6. Major Contributions

This section summarizes the major contributions and achievements gained from the research carried out on this work. The order in which they are listed does not express any relative importance.

6.1. *Identification of the lack of flexibility in component models for system software*

An important contribution of this research is the identification of the lack of flexibility in existing component models for systems software. It is now well identified that the existing technologies are inadequate to construct systems that span target domains and deployment environments. In addition, it hinders component reuse since each platform has its own component style and implementation.

Several component models have been briefly discussed in Sec. 3 primarily to demonstrate the inefficiency of the existing component models to build systems software for a variety of system domain. Generally, the state-of-the-art has ignored the deployment environment heterogeneity and has focused primarily on solving a specific challenge. In our case studies, we have shown that primitive and complex component implementations (such as the NetBind components) are deployable through OpenComL without any change on the deployed components.

6.2. *A globally-applied component model*

An important contribution of this work is the provision of a globally-applied component model promoted by OpenComL. OpenComL relies on a unique approach that is ubiquitously applied in all deployment environments including the low-level ones, such as the microengines in the IXP1200 router environment. This is well illustrated in the case study that was presented in Sec. 5.1. As seen in this case study, components are deployed (using the appropriate caplets, loaders and binders) in a standard and uniform way across multiple deployment environments that exist in this router platform.

This facilitates programming primarily heterogeneous environments such as an NP-based router. As reported previously, the developer merely sees components and the corresponding extension components (i.e. caplets, loaders and binders) that are deployed in a uniform approach.

6.3. *OpenCom comprehensive evaluation*

The OpenComL component model follows the OpenCom approach, an OS-independent component-based building system technology. This work contributed a comprehensive instantiation and evaluation of the OpenCom model. This was achieved by instantiating the OpenCom approach to provide a component model for Linux environments. The OpenComL lets us create systems targeted to a variety of domains and environments where Linux is supported. This is demonstrated by two case studies that rely on OpenComL to build complex and non-trivial systems in a highly heterogeneous platform such as an NP-based router.

The proposed extensions framework along with loader and binder components help us abstract and implement individual deployment mechanisms for each environment. Component reusability is promoted, given that existing components become deployable as long as the required loader and binder components are plugged into the extensions framework. This was the case for NetBind components (in the first case study) where a particular loader and binder extension makes Microcode components deployable in the IXP1200 router environment.

The contribution also comprehended the use of OpenComL to construct parallel software based on the CSP paradigm. FlexPar was evaluated in Linux environments and provides a unique tool to construct software based on a variety of CSP-based languages. These languages are often acknowledged as rigid, and thus the OpenComL flexibility can alleviate and foster the use of such a paradigm. For example, developers may be unaware of the complexity of each CSP-based language thanks to the loader and binder extensions.

7. Conclusions and Further Work

This paper contributed with OpenComL, a generic form of technology for building component-based systems software in Linux environments. We presented the major design features of our technology, discussed its implementation and the use of OpenComL in constructing two complex and non-trivial systems. Given that Linux runs in a variety of deployment environments ranging from mainstream PCs to sensor nodes, it demonstrated to be well-suited to validate the generality and tailorability of OpenComL in such a environment. We discussed that the generality can bring a number of benefits including skill-sets transference across several technologies and that it also helps us to ensure that a single tool is employed while attempting to build software targeted to a variety of environments and system domains. Finally, OpenComL has been applied to the needs of both present and future work. For example, we are investigating the use of OpenComL to provide a reconfigurable fault-tolerance mechanism specially targeted to real-time systems running on embedded devices. This will potentially allows us to apply individual fault tolerance and recovery policies to a particular environment and system domain.

Acknowledgments

The first and second author would like to thank FAPESP for funding the bulk of this research project (Ref. 2006/06576-8 and Ref. 2008/05346-4). The first author would also like to thank the Brazilian Research Council (CNPq) for funding the REDE research project (Proc. 474803/2009-0).

References

1. E. Bruneton, T. Coupaye and J. Stefani, Recursive and dynamic software composition with sharing, in *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming* (*WCOP'02*), 2002.
2. A. T. Campbell, M. E. Kounavis, D. A. Villela, J. B. Vicente, H. G. de Meer, K. Miki and K. S. Kalaichelvan, NetBind: A binding tool for constructing data paths in network processor-based routers, in *5th IEEE International Conference on Open Architectures and Network Programming* (*OPENARCH'02*), June 2002.
3. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama and T. Sivaharan, A generic component model for building systems software, *ACM Transactions on Computer Systems* **26**(1) (2008).
4. M. Debbage, Southampton's portable occam compiler(SPoC), 1994.
5. J. Dowling and V. Cahill, The K-component architecture meta-model for self-adaptive software, In *3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (*REFLECTION'01*), Kyoto, Japan, September 2001, LNCS Vol. 2192, pp. 81–88.
6. J. P. Fassino, J. B. Stefani, J. Lawall and G. Muller, THINK: A software framework for component-based operating system kernels, in *USENIX 2002 Annual Conference*, June 2002.
7. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers, The flux OSKit: A substrate for kernel and language research, in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (ACM Press, 1997), pp. 38–51.
8. H. Hansson, M. Akerholm, I. Crnkovic and M. Torngren, SaveCCM — A component model for safety-critical real-time systems, in *Euromicro Conference, Special Session Component Models for Dependable Systems*. IEEE, September 2004.
9. H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg and J. Wolter, The performance of μ-kernel-based systems, in *16th ACM Symposium on Operating Systems Principles*, October 1997, pp. 66–77.
10. J. Helander and A. Forin, MMLite: A highly componentized system architecture, in *8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998, pp. 96–103.
11. C. A. R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, 1985).
12. Intel, Intel IXP1200. http://www.intel.com, 2002.
13. Christian L. Jacobsen and Matthew C. Jadud, The Transterpreter: A Transputer Interpreter, in Dr. Ian R. East, Prof. David Duce, Dr. Mark Green, Jeremy M. R. Martin and Prof. Peter H. Welch (eds.), *Communicating Process Architectures 2004*, *Concurrent Systems Engineering Series* **62** (2004) 99–106.
14. S. Karlin and L. Peterson, VERA: An extensible router architecture, in *4th International Conference on Open Architectures and Network Programming* (*OPENARCH*), April 2001.
15. Inmos Limited, *Occam2 Reference Manual* (Prentice Hall, 1984).
16. Linux Online, Linux Online — Shared Libraries, http://www.linux.org/, 2005.

17. P. Maes, Concepts and Experiments in Computational Reflection, in *Proceedings of the Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA'87*), ACM, October 1987, pp. 147–155.
18. Microsoft Corporation, Net Home Page, http://www.microsoft.com/net, 2001.
19. R. Morris, E. Kohler, J. Jannoti and M. Kaashoek, The click modular router, in *17th ACM Symposium on Operating Systems Principles* (*SOSP'99*), Charleston, SC, USA, December 1999.
20. Mozilla Organization, XPCOM Project, http://www.mozilla.org/projects/xpcom, 2001.
21. J. Muskens and M. Chaudron, Prediction of run-time resource consumption in multi-task component-based software systems, in *Proceedings of the 7th International Symposium on Component-Based Software Engineering* (*CBSE7*), Edinburgh, Scotland, May 2004.
22. OMG, Object Management Group, Interface Definition Language (IDL). http://www.omg.org/cgi-bin/doc?formal/02-06-07.
23. OMG, The CORBA Component Model. orbos/99-07-01.
24. R. Ommering, F. Linden, J. Kramer and J. Magee, The Koala component model for consumer electronics software, *j-COMPUTER* **33**(3)(2000)78–85.
25. P. H. Welch and J. M. R. Martin, A CSP model for Java multithreading, in P. Nixon and I. Ritchie (eds.), *Software Engineering for Parallel and Distributed Systems* (*ICSE 2000*) (IEEE Computer Society Press, 2000), pp. 114–122.
26. J. Polakovic, S. Mazare, J.-B. Stefani and P.-C. David, Experience with safe dynamic reconfigurations in component-based embedded systems, in *10th International SIGSOFT Symposium on Component-Based Software Engineering* (*CBSE 2007*), 2007, pp. 242–257.
27. M. Roman and N. Islam, Dynamically programmable and reconfigurable middleware services, in *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, New York, NY, 2004, pp. 372–396.
28. N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb, S. Karlin, A. Nakao, X. Qie, T. Spalink and M. Wawrzoniak, Extensible routers for active networks, in *DARPA AN Conference and Exposition* (2002), pp. 92–116.
29. B. C. Smith, *Reflection and Semantics in a Procedural Language*, PhD thesis, Massachusetts Institute of Technology, January 1982.
30. Sun Microsystems, Enterprise Java Beans Specification Version 1.1, http://java.sun.com/products/ejb/index.html.
31. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. (Addison-Wesley, 2002).
32. Telos sensor motes, Crossbow. http://www.xbow.com/Products/Wireless_Sensor_Networks.htm, 2005.
33. J. Ueyama, *A Runtime Component Model for System Software*, PhD thesis, Lancaster University, August 2006.
34. J. Ueyama, E. R. M. Madeira and P. Grace, FlexPar: A reconfigurable middleware for parallel environments, in *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing* (*ISORC 2008*), Orlando, Florida, USA, May 2008.
35. P. H. Welch and F. R. M. Barnes, Communicating Mobile Processes: Introducing occam-pi, in A. E. Abdallah, C. B. Jones and J. W. Sanders (eds.), *25 Years of CSP*, *Lecture Notes in Computer Science*, Vol. 3525 (Springer Verlag, 2005), pp. 175–210.

36. P. H. Welch and D. C. Wood, The kent retargetable occam compiler, in Brian O'Neill (ed.), *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering* (World occam and Transputer User Group, IOS Press, 1996), pp. 143–166.
37. M. Winter, T. Genbler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, P. Muller, C. Stich and B. Schonhage, Components for embedded software: The PECOS Approach, in *CASES '02: Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, New York, NY, USA, 2002, pp. 19–26.