Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

K-CSP: Component Based Development of Kernel Extensions

Bernhard SPUTH

DSP Centre, Ngee Ann Polytechnic, Block 8 #06-09,
Clementi Road 535, 599489 Singapore, Singapore
and
Department of Engineering, University of Aberdeen, Aberdeen, AB24 3UE, UK
bernhard@erg.abdn.ac.uk

Alastair R. ALLEN

Departments of Engineering and Bio-Medical Physics,
University of Aberdeen, Aberdeen, UK

Abstract. Kernel extension development suffers from two problems. Firstly, there is little to no code reuse. This is caused by the fact that most kernel extensions are coded in the C programming language. This language only allows code reuse either by using ‘copy and paste’ or by using libraries. Secondly, the poor separation of synchronisation and functionality code makes it difficult to change one without affecting the other. It is, therefore, difficult to use the synchronisation mechanisms correctly. The approach proposed in this paper tries to solve these problems by introducing a component based programming model for kernel extensions, and a system based on this proposal is implemented for the Linux kernel. The language used for the implementation is Objective-C, and as a synchronisation mechanism Communicating Sequential Processes is used. This model allows the functionality and synchronisation of a component to be developed separately. Furthermore, due to the use of Communicating Sequential Processes it is possible to verify the correctness of the synchronisation. An example given in this paper illustrates how easy it is to use the K-CSP environment for development.

1 Introduction

K-CSP is an extension to the Linux Kernel, which brings Component Based Programming into the domain of kernel extension development. K-CSP provides a proven synchronisation model in the form of Communicating Sequential Processes (CSP)[1]. Objective-C [2] is used as an object oriented environment to allow easy code reuse and therefore to increase code quality. The K-CSP architecture is generally applicable to any operating system based upon the C programming language.

Kernel extensions are runtime loadable modules which enhance the functionality of an operating system (OS) kernel. They are used to provide device drivers for new hardware or new file systems, without recompiling the kernel [3]. Once a module is loaded it becomes part of the kernel. A defective kernel extension may bring down the kernel and with it all other running programs, possibly resulting in data loss on the user’s side. Therefore, it is necessary to develop kernel extensions in a defensive way. Defensive programming is characterised by performing consistency checks on variables before use. For instance, checking for null pointers in passed function parameters. This avoids errors caused by dereferencing such a pointer. A single null pointer dereference can cause mayhem in the running kernel.

Unfortunately, the kernel environment is not easy to work in. First of all, the kernel is a concurrent system with the ability to utilise multiple processors. This requires that kernel extensions have to be multiprocessor safe, by default. The kernel environment provides basic synchronisation mechanisms for this task, like spinlocks and semaphores. These synchronisation mechanisms must be used to guard critical code sequences in order to avoid concurrent access. Improper use of synchronisation mechanisms can cause deadlocks. Especially for an inexperienced developer, it is hard to judge whether a code sequence needs to be guarded or not. Another difficulty comes from the fact that an error in the synchronisation does not necessarily show up immediately. Even if it does, it may be hard to reproduce. Errors which cannot be reproduced are hard to fix, as there is no way to test whether the error is removed or not.

The next problem with classical synchronisation mechanisms is that functionality and synchronisation are mixed within the code. This makes it difficult to modify one without affecting the other. Especially if the code is modified at a later time, a previous working synchronisation might be broken. To avoid these problems, a synchronisation mechanism is required that allows the separation of synchronisation and functionality.

A first step towards this separation is access limitation. This is achieved by data hiding, a technique propagated by object oriented programming (OOP)[4]. In OOP, resources are contained within objects. Each resource has an access limiter assigned to it, allowing access from outside the object (unlimited) or not (limited). To access a resource with limited access, a method of the object needs to be called. This method performs all necessary synchronisation operations. Such methods can be used to synchronise access to resources. This soothes the synchronisation problems, but on the other hand it requires an object oriented environment in the kernel. An example of such an operating system is Apertos [5, 6, 7] which is object oriented from the ground up. This is great in terms of easy code reuse, which generally results in shorter development time and increasing the quality of the code. Unfortunately, the development of kernel extensions is limited in Apertos, because only a single thread of execution is allowed for each extension. While this is good for avoiding synchronisation problems, for some types of hardware it is necessary to have a second thread polling the device. This is, for instance, the case in kernel extensions using the isochronous stream of USB devices [8].

In order to support this type of hardware, an architecture is required that allows multiple threads of execution within a single kernel extension. Therefore, this approach needs to provide a means of abstraction for the packaging and synchronisation of threads. To encourage code reuse, it should be possible to create new kernel extensions out of pieces of already existing kernel extensions. These pieces of kernel extensions are so called *components* [9], and this is what Component Based Programming is about. The example given later in this document shows how easy it is with K-CSP to create components and use them. It also demonstrates the ease of developing multi-threaded kernel extensions with K-CSP.

2 Component Based Programming for Kernel Extensions

The K-CSP architecture attempts to bring component based programming into the domain of kernel extension development. This section gives an introduction to component based programming as well as traditional kernel extension development.

2.1 *What is Component Based Programming?*

Component Based Programming (CBP) is a programming methodology which focuses on easy and secure code reuse. It is the next logical step after Object Oriented Programming (OOP). The core idea is to build systems out of already existing components, instead of constantly *reinventing the wheel*. A component itself is a piece of self contained code providing a functionality, able to communicate with other components.

Component Based Programming is widely used in the field of User Interface development: examples are Desktop JavaBeans from Sun [10] or ActiveX Controls by Microsoft [11]. Components are also used in the field of distributed programming, for instance in Enterprise JavaBeans [12], or the CORBA Component Model [13].

Systems developed with CBP are more clearly structured and less error prone. CBP consists mainly of assembling components, rather than implementing functionality. This results in shifting the work of a developer from coding a hand tailored solution to designing a solution out of already available components. As each component only solves one problem, its size is most probably within the sweet spot of around 200 lines of code, where defect density is lowest according to Hatton [14]. Programs created with CBP are similar to block diagrams. This allows easier understanding of the programs, making them easy to debug, extend and modify.

2.2 Kernel Extension Development for the Linux Kernel

The Linux Kernel has been implemented using the C programming language [15]. Kernel extensions become part of the kernel once they are loaded. To be able to interface to the kernel, extensions need to comply with the kernel environment. This requires them to conform to the kernel calling conventions. Calling conventions are defined by the programming language. Therefore, kernel extensions are traditionally developed using the C programming language.

2.2.1 C Language Restrictions for CBP

The C programming language is a procedural programming language. Therefore, the highest layer of abstraction is a procedure or function. Data hiding above the function level is not supported by the C programming language. Free use of global variables is common and results, most of the time, in mayhem, when global variables are manipulated unexpectedly. This makes C unsuitable for a component based system. A component is self-contained and consists of code and data which must not be tampered with from outside the component.

Another point to consider is the lack of concurrency support by the C programming language. There are no language constructs for synchronisation. With no standardisation by the language, each OS provides its own synchronisation model. This makes it difficult to port kernel extensions, especially device drivers form one OS to another.

2.3 Comparison of Component Based Programming and Traditional Kernel Extension Programming

In the traditional way of developing kernel extensions, everything had to be implemented and tested from the ground up. This resulted in unstructured, error prone code. When using a component based approach, a programmer can rely on previously developed and tested components. The programmer's work changes from coding his own solution, to designing a solution out of already existing components. This hopefully increases the code quality of the solution. Therefore, a programmer becomes more an engineer than an artist. As components need to be created before they can be used, the component architecture needs to allow easy implementation and testing.

3 K-CSP Architecture

The aim of K-CSP is to bring component based programming into the Linux Kernel domain. Therefore, K-CSP has to become a part of the Linux Kernel, with all the previously mentioned limitations.

In order to develop an environment for component based programming the following requirements have to be met:

- Components need to be implementable in a self contained fashion. They should only allow interaction over fixed interfaces.
- Components must be able to be executed concurrently.
- There must be a safe way to connect components together.

In K-CSP, the function and the synchronisation have to be separated. This will avoid the chaos caused when changing one affects the other.

3.1 Components as Self-Contained Code

As component based programming is an evolutionary step from object oriented programming, it is sensible to base a component environment on an object oriented (OO) environment. Unfortunately, we cannot use C++ as a base, because the C++ namespace collides with the kernel headers. To use C++, the Linux Kernel would have to be reviewed and partly rewritten, according to the Linux Kernel FAQ [16]. In order to be able to use an OO-environment, we have two choices. We could develop our own OO-environment. The downside of this is that it will require a lot of work, and the solution is not reliable, because it is completely new. A new OO-environment would require the users to learn a new programming language. The second option we have is to take an already existing object oriented environment that conforms to the C conventions and port it into the kernel. Objective-C [2] is such an environment. Objective-C has the advantage that the code is translated into C before being compiled into a binary. This makes it possible to create kernel extensions which fully comply with the set of kernel requirements and at the same time have the data hiding capabilities of an object oriented environment.

3.2 Concurrent Execution of Components

Components need to react to requests from outside and inside. Outside requests, for a device driver, are for instance issued by the device. An example of such a request is an interrupt, which is handled by calling the interrupt service routine (ISR). This ISR is then executed concurrently with the remainder of the device driver. Requests coming from the user side are handled similarly. These components are implementable without using a thread: they are run using the thread of the caller. As they are used as an interface to the outside world of the kernel extension, they are called *Interfacing Components* (IC). These components do not execute in a recursive way.

Components that handle only internal requests are Control Components (CC). Control components perform monitoring or polling tasks. These components need to run recursively and therefore, they are executed in their own thread.

Both component types IC and CC can be modelled as CSP processes.

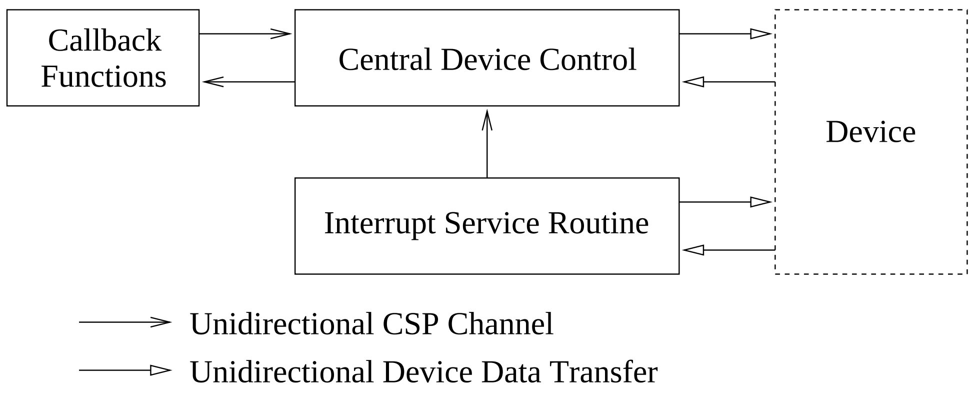
Callback Functions
Central Device Control
Device
Interrupt Service Routine
Unidirectional CSP Channel
Unidirectional Device Data Transfer

Figure 1: Example of a Device Driver implemented with K-CSP

3.3 Component Communication Infrastructure

With the components being executed concurrently, the communication between them has to be multi-thread safe, and for multiprocessor machines also multiprocessor safe. The channel construct introduced by CSP takes care of these problems and is intuitive to use. The component communication infrastructure is therefore implemented in the form of CSP Channels.

3.4 Resulting Model

The K-CSP architecture relies upon Objective-C to provide an object oriented environment, using a runtime library. In order to allow secure concurrency within K-CSP, a CSP subsystem will be created, which itself is based on the Objective-C runtime. With CSP as the synchronisation mechanism it is possible to prove the correct implementation of the synchronisation. The process and communication structure of a Device Driver implemented in K-CSP is shown in Figure 1. The figure gives a good abstraction of the different components and their interactions, like a block diagram.

4 Implementation Aspects of K-CSP

K-CSP is based upon Linux Kernel 2.6.3 [17]. GNU GCC version 3.3.x supplies the basis of the Objective-C runtime. The interface to the CSP subsystem is similar to that of JCSP [18].

4.1 Bringing Objective-C into the Kernel

The Linux Kernel is usually compiled using the C compiler of the GNU Compiler Collection (GCC) [19]. GCC also includes an Objective-C compiler. Therefore, the GNU *Objective-C Runtime Library* (runtime) can be used. But even with the runtime already available there are still a number of problems to solve:

- The runtime is meant to run in user mode. Therefore, it is necessary to port the code into kernel mode and resolve any incompatibilities.
- The Objective-C compiler utilises special segments in the ELF binary format [20], in order to register the classes of the program with the runtime. These segments are not supported by the module loader supplied by the Linux kernel.

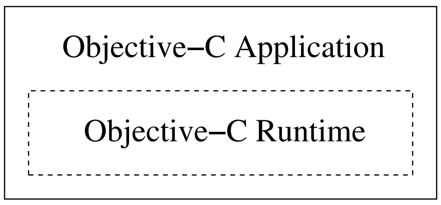

Figure 2: Tight coupling of Application and Objective-C Runtime

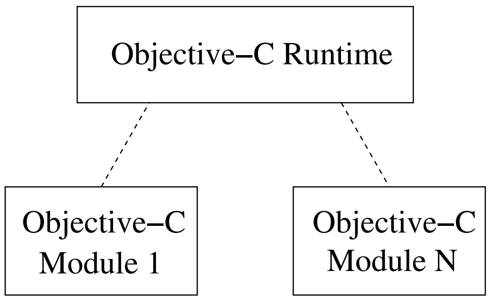

Figure 3: Loose coupling of Modules and Objective-C Runtime

- The Objective-C runtime is only meant to be used by one program, as shown in Figure 2. The runtime becomes part of the application and is terminated together with it. Therefore, the runtime does not support modules unregistering their classes when being unloaded. In the K-CSP implementation it is necessary to have multiple modules sharing the Objective-C runtime to enable usage of classes defined in other modules. This results in a loose coupling of Objective-C runtime and Modules, as illustrated in Figure 3. The kernel allows unloading of modules, if they are not in use. When unloaded, entries of these modules are still in the runtime registry and the next time a module tries to register entries of the same name the old entries are used. This is, for instance, the case when unloading and reloading a module. Loading and unloading of modules is not only a common practice during module development, but also in environments where hardware is connected to or disconnected from the system during runtime. The entries of the runtime registry contain pointers to the methods and members of the classes. The pointers must not stay valid when a module gets reloaded: if now an old entry is used it might refer the wrong location, resulting in a segmentation fault. To solve this problem entries of a module need to be purged from the runtime registry when a module gets unloaded.

4.1.1 Porting the Objective-C Runtime into the Linux Kernel

The availability of the GNU GCC Objective-C runtime in source code made the porting possible. The source code was developed in a modular way, encapsulating platform specific issues. All this made the porting a painless process. The most labour intensive task was to export the functions to the kernel, in order to make them accessible from outside the module.

4.1.2 Loading of Objective-C Kernel Extensions

The Linux Kernel is compiled in the ELF binary format [20], which is also used for kernel extensions. Objective-C programs are compiled in this file format. A binary complying with the ELF binary format consists also of multiple sections, each with a special purpose. The Objective-C compiler utilises the *.ctors* section (constructors) in order to register the classes contained in the binary with the runtime library. The routines specified in the *.ctors* segment are executed when loading the binary, before the main entry point is executed. For normal C binaries this is not necessary, therefore the module loader of the Linux Kernel, by default, ignores this segment. The kernel module loader had to be modified to execute available entries of the *.ctors* segment. With this modification it is possible to load kernel extensions developed with Objective-C with the Linux Kernel.

In order to execute a kernel extension it is necessary to supply a program entry point. For the Linux Kernel this entry point is defined to be a C function. Unfortunately, it is not possible to use a method of an object as entry point. This is due to the fact that an object first needs to be allocated and initialised, before a method of it can be called. Furthermore, methods in Objective-C get passed a hidden parameter, the *self* pointer. The *self* pointer is similar to the *this* pointer of C++ [21]. This automatically makes Objective-C methods unsuitable for callback functions of the Linux Kernel, because their fixed interface does not include the *self* pointer. This is the reason why it is necessary to provide wrapper functions, which get registered as callback functions and relay the parameters to the methods of an object.

4.1.3 Unloading of Modules

The original Objective-C runtime was designed to be used only by one program at a time. This is the reason why there was no concept of unloading. It is necessary to remove the previously registered entries from the Objective-C runtime, because the Linux Kernel requires the unloading functionality. So the runtime was extended with a function to remove the class definition for a specific class. This function needs to be parameterised, in order to unload the correct class definition. Therefore, each class of an Objective-C kernel extension has to come with its own call to the unload function. Removing a class definition while the objects of that class are still in use, results in the objects not working correctly. It is therefore of importance to only unload class definitions not in use. Assuming that no objects are in use after the execution of the module exit function by the kernel module loader, the module loader can safely execute the class definition unload functions provided by the module. To do so, these unload functions are registered as destructors in the *.dtors* section of the ELF binary format. This allows them to be found as long as the kernel extension is loaded.

After all these modifications to the Linux Kernel and the Objective-C runtime, we are now able to use Objective-C for development of Linux Kernel Extensions. This is the object oriented base upon which K-CSP is built.

4.2 CSP Subsystem Implementation

The CSP subsystem implementation is built on top of the Objective-C runtime. This allowed implementation of the CSP subsystem in an object oriented fashion and also aided as a test of the Objective-C runtime kernel port. Furthermore, this allows the porting of the CSP implementation to other OS where the Objective-C runtime is also available. At the moment the CSP subsystem supports the following CSP constructs:

- Process: In the K-CSP environment a Process is created by implementing the CSProcess protocol.

- Channel: The channel construct is only implemented in the form of a point to point channel (One2OneChannel). It supports alternation of channel inputs. Multi-point channels and call channels will follow soon.

- Alternative: The alternative construct only supports fair alternation selection methods. Extensions to priority selections are planned.

- Parallel: This is the normal parallel construct. It is used to implement a Process Network construct as available in JCSP.

5 Example of Utilising K-CSP

The ease of use of K-CSP is demonstrated in this section. The example is very simple, as it only shows one process sending messages to another process. Implementing something similar as a kernel extension without K-CSP would have required at least double the amount of code. The example given shows the power of Objective-C and CSP combined. This example demonstrates how the runtime type information system of the Objective-C environment can be used to terminate a process network, by sending poison messages. Before going into the example some background information on Objective-C and the K-CSP API is given.

5.1 Objective-C Background

In Objective-C all classes are derived from the class Object. As with Java, only single inheritance is allowed, but a class can implement multiple protocols. A protocol is the Objective-C version of an interface in Java. Objective-C separates the declaration of a class and its implementation. The declarations are stored in header files, just as in C/C++. The implementation is given in files with the extension 'm', called m-files. In the listing shown in this section, declaration and implementation are combined.

Every object in Objective-C provides methods for Runtime Type Information (RTTI)[22]. A few of these mechanisms are used in this example:

- `(const char *) name`
  This method returns the name of the class of this object.

- `(BOOL) isKindOf:` *`class-object`*
  Will return *YES* when the object is of type *`class-object`* or a descendant.

- `(BOOL) conformsTo:` *`protocol`*
  Returns *YES* if the object implements *`protocol`*.

5.2 K-CSP API

This example utilises the CSP subsystem of K-CSP. To be able to understand the code snippets, some information is given below on the API of the K-CSP elements used.

- `CSProcess`
  This protocol defines how processes have to be implemented.

- `One2OneChannel`
  This class provides a channel having one input and one output end. For interaction with processes two methods are provided.

  – `(BOOL) read: (id*)` *`pMessage`*
    This method is used by the receiver to read a message from the channel. This method returns a value of type BOOL, which can either be *YES* or *NO*. The method will return *NO* if an error has occurred. A reference to the message is passed to the *`pMessage`* parameter.

  – `(BOOL) write: (id)` *`message`*
    This method is used by the sender to pass a message to the receiving process. The boolean return value of the method indicates whether an error has occurred.

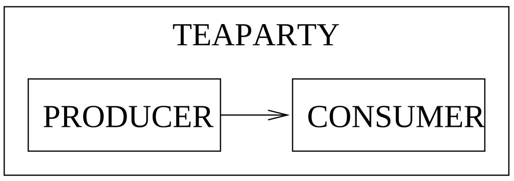

Figure 4: The TEAPARTY process network

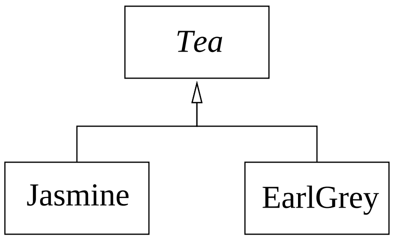

Figure 5: Class tree for the different types of Tea

- `Parallel`
  The Parallel class provides a construct used to execute multiple processes in parallel.
  - `(BOOL) add: (id)` *`object`*
    The add method is used to add a process to an object of the Parallel class. The return value indicates whether the method completed successfully.
  - `(BOOL) run`
    The run method executes all processes added to the parallel object. It returns when all processes have finished execution. The return value of the run method is *YES* if no errors were encountered.

5.3 CSP Subsystem Internals

The CSP subsystem of K-CSP is designed in the form of a kernel extension. It is implemented in the Objective-C programming language. In K-CSP each created process is given its own thread using operating system functions. This allows the OS to distribute processes upon available CPUs. Operating system supplied synchronisation mechanisms are used for the implementation of Parallel, One2OneChannel and Alternative. The implementation of the CSP subsystem of K-CSP, shows that Objective-C indeed embeds nicely in a kernel environment designed for C.

5.4 Tea Party Example

This Tea Party example consists of two processes, PRODUCER and CONSUMER, which are connected using a One2OneChannel. They are combined to create a third process, called TEAPARTY. Figure 4 illustrates the resulting process network. The PRODUCER sends messages to the CONSUMER. The CONSUMER analyses the incoming messages. If they are objects implementing the Tea protocol, a text message is printed onto the standard output. If the message is of type Poison the CONSUMER terminates. The class tree for Tea is shown in Figure 5.

5.4.1 PRODUCER

The PRODUCER sends three messages, first EarlGrey, then Jasmine and finally Poison after which it terminates. Equation 1, shows the CSP version of the PRODUCER process. The implementation in K-CSP is shown in Listing 2. The translation of CSP into K-CSP has to be done manually for the time being.

$$PRODUCER = x!EarlGrey \rightarrow x!Jasmine \rightarrow x!Poison \rightarrow STOP \tag{1}$$

Listing 1: K-CSP version of the CONSUMER

```
@interface CONSUMER: Object <CSProcess>
{
  id<ChannelInput> In;
}
-(id) initWithChannelInput: (id) ChannelInput;
@end

@implementation CONSUMER
-(id) initWithChannelInput: (id) ChannelInput;
{
  [super init];
  In = ChannelInput;
  return self;
}
-(BOOL) run
{
  id message;
  while(1){
    if(NO == [In read: &message])
    {
      dbgprint("Received Terminating Signal \n");
      return NO;
    }
    dbgprint("CONSUMER received a message of type: \
      %s\n",[message name]);
    // checking whether this message is poisonous
    if([message isKindOf: [Poison class]] == YES){
      dbgprint("CONSUMER received poison, now terminating\n");
      //releasing the message object.
      [message free];
      return YES;
    }
    // checking if the message implements the Tea protocol
    if([message conformsTo:@protocol(Tea)] == YES){
      dbgprint("Received a cup of Tea, thank You\n");
    }
    // releasing the message object
    [message free];
  }
}
@end
KOBJC_CLASS(CONSUMER);
```

Listing 2: K-CSP version of the PRODUCER process

```
@interface PRODUCER: Object <CSProcess>
{
  id<ChannelOutput> Out;
}
-(id) initWithChannelOutput: (id) ChannelOutput;
@end

@implementation PRODUCER
-(id) initWithChannelOutput: (id) ChannelOutput;
{
  [super init];
  Out = ChannelOutput;
  return self;
}
-(BOOL) run
{
  // allocating, initialising and sending an EarlGrey
  // object over the channel
  if(NO == [Out write: [[EarlGrey alloc] init]])
  {
    dbgprint("Received Signal terminating\n");
    return NO;
  }
  if(NO == [Out write: [[Jasmine alloc] init]])
  {
    dbgprint("Received Signal terminating\n");
    return NO;
  }
  if(NO == [Out write: [[Poison alloc] init]])
  {
    dbgprint("Received Signal terminating\n");
    return NO;
  }
  return YES;
}
@end
//This macro creates the required unload function to
//remove the entries of the class PRODUCER from the
//runtime registry
KOBJC_CLASS(PRODUCER);
```

5.4.2 CONSUMER

The CONSUMER process analyses every incoming message, to decide whether it is poisonous or not. If it is not poisonous, the message is further examined to determine if it is of type *Tea*, in which case a greeting will be printed. After printing the greeting the process will again wait for an incoming message. If the message received is poisonous, the process will immediately terminate. The CONSUMER CSP representation is given in Equation 2, with the K-CSP implementation in Listing 1.

$$\begin{aligned} CONSUMER &= x?message : \{Tea, Poison\} \to P(message) \\ \text{where} & \\ P(Tea) &= printGreeting \to CONSUMER \\ P(Poison) &= STOP \\ \text{with} & \\ Tea &= \{EarlGrey, Jasmine\} \end{aligned} \tag{2}$$

5.4.3 TEAPARTY Process

The TEAPARTY process provides the environment for the ongoing tea party. Its main task is to interconnect the PRODUCER and CONSUMER processes, using a One2OneChannel and executing them in parallel. After the execution of the processes is completed the tea party is over and the TEAPARTY process terminates.

The CSP representation is given in Equation 3. The implementation using the K-CSP environment is shown in Listing 3. The console output of the TEAPARTY process is shown in Listing 4.

$$TEAPARTY = PRODUCER \parallel CONSUMER \tag{3}$$

Listing 3: K-CSP version of the TEAPARTY process network

```
void TEAPARTY(void){
  // Allocating and initialising a One2OneChannel object.
  One2OneChannel *chan = [[One2OneChannel alloc] init];
  // Allocating and initialising the PRODUCER and CONSUMER process objects.
  PRODUCER *pro = [[PRODUCER alloc] initWithChannelOutput: chan];
  CONSUMER *con = [[CONSUMER alloc] initWithChannelInput: chan];
  // Allocating and initialising an object of the class Parallel
  Parallel *par = [[Parallel alloc] init];

  // Adding the PRODUCER and CONSUMER objects to the parallel object.
  [par add: pro];
  [par add: con];
  // Executing the processes using parallel.
  if(NO == [par run]){
    dbgprint("TEAPARTY_run something is fishy\n");
  }
  // cleaning up after the tea party, freeing all objects.
  [par free];
  [con free];
  [pro free];
  [chan free];
  return;
}
```

Listing 4: Output of the Tea Party onto the console

```
CONSUMER received a message of type: EarlGrey
Received a cup of Tea, thank You
CONSUMER received a message of type: Jasmine
Received a cup of Tea, thank You
CONSUMER received a message of type: Poison
CONSUMER received poison, now terminating
```

5.5 Comparing the K-CSP Implementation with a Traditional Implementation

The total time to implement the example was around two hours. The implementation of a kernel extension with similar functionality would have taken at least one day and resulted in a large amount of code. This is caused by the difficulty of creating and destroying threads in the kernel. Especially the secure stopping of a thread is not easy. But in K-CSP the programmer does not need to bother about it, which allows easier application of multithreading when necessary. Furthermore, due to the use of CSP channels it is possible to safely exchange data between the threads.

6 Conclusions

This paper gave an introduction to K-CSP, a component architecture for Linux kernel extensions. While K-CSP is still not fully developed, it is clear that the proposed model can make kernel extension development easier, faster and less error prone. The example given showed how simple it is with K-CSP to create components and use them. Due to the use of Objective-C as the object oriented environment and CSP as synchronisation mechanism, K-CSP can be easily ported to other Operating Systems which also use C as the development language. Such operating systems are: all linux flavors, FreeBSD, OpenBSD, Mac OS-X (based on BSD), Windows NT and its successors. This could lead to a simplification of kernel extension development, due to similar programming environments.

7 Further Work

With K-CSP still under development there are a lot of points still to be addressed. Of course an environment for component based programming must come with a set of components for the most common tasks. This enables programmers to benefit from CBP immediately. For programmers wanting to develop components for K-CSP, the components in this paper will act as examples. The CSP subsystem at the moment only comes with a limited set of CSP constructs: the number of supported constructs should be enlarged. An automatic translator from CSP to K-CSP, similar to that presented by G.S. Stiles in [23], could help to further increase code quality while decreasing development time. In order to avoid memory holes, the inclusion of a Garbage Collector (GC) would be appropriate. The Objective-C runtime comes with support for GC through an external library. The Linux Kernel build system at the moment does not support the use of the Objective-C compiler directly, which results in inserting calls to the Objective-C compiler in the makefiles. This could be avoided by enabling the build system to accept Objective-C files directly and using the Objective-C compiler. This is a point of enhancement for the convenience of the programmer.

References

[1] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[2] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley Pub Co, 2nd edition, May 1991. ISBN: 0201548348.

[3] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers: Second Edition*. O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, 2nd edition, June 2001. 0-596-00008-1.

[4] Grady Booch. *Object Oriented analysis and design*. Addison Wesley Longman Inc., One Jacob Way, Reading, Massachusetts 01867 USA, 1993.

[5] Jun-ichiro Itoh and Yasuhiko Yokote. Concurrent object-oriented device driver programming in apertos operating system. Technical report, Sony Computer Science Laboratory, Keio University Department of Computer Science, August 1994.

[6] Jun-ichiro Itoh, Yasuhiko Yokote, and Mario Tokoro. SCONE: Using concurrent objects for low-level operating system programming. Technical report, Sony Computer Science Laboratory, Keio University Department of Computer Science, March 1995.

[7] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. *ACM SIGPLAN Notices*, 27(10):414–434, 1992.

[8] Jan Axelson. *USB Complete: Everything you need to develop custom USB Peripherals*. Lakeview Research, 2209 Winnebago, St. Madison, WI 53704 USA, 2nd edition, 1999. ISBN: 0-9650819-3-1.

[9] Ju An Wang. Towards component-based software engineering. In *Proceedings of the eighth annual consortium on Computing in Small Colleges Rocky Mountain conference*, pages 177–189. The Consortium for Computing in Small Colleges, 2000.

[10] Desktop Java JavaBeans. Sun JavaBeans Website. `http://java.sun.com/products/javabeans/`.

[11] Microsoft COM technologies - information and resources for the component object model-based technologies. Website. `http://www.microsoft.com/com/`.

[12] Enterprise JavaBeans specification, version 2.1. Specification published by SUN Microsystems, November 2003. Version 2.1, Final Release. `http://java.sun.com/products/ejb/docs.html`.

[13] CORBA Component Model, v3.0. Specification of the Object Management Group, June 2002. `http://www.omg.org/technology/documents/formal/components.htm`.

[14] Les Hatton. Reexamining the fault density – component size connection. *IEEE Software*, 14(2):89–97, March/April 1997.

[15] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall PTR, Upper Saddle River, NJ 07458, USA, 2nd edition, March 1988. ISBN: 0131103628.

[16] The Linux kernel mailing list FAQ. Internet. `http://www.tux.org/lkml/\#s15-3`.

[17] The Linux kernel archives. Internet. `http://www.kernel.org/`.

[18] Communicating Sequential Processes for Java (JCSP). Internet. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[19] GCC home page - GNU project - Free Software Foundation (FSF). Internet. `http://gcc.gnu.org/`.

[20] Hongjiu Lu. Elf: From the programmer's perspective. Technical report, NYNEX Science & Technology Inc., 500 Westchester Avenue, White Plains, NY 10604, USA, May 1995.

[21] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman Inc., One Jacob Way, Reading, Massachusetts 01867 USA, special edition, March 2000.

[22] Stephen G. Kochan. *Programming in Objective-C*. Sams Publishing, 800 East 96th Street, Indianapolis, Indiana 46240, USA, first edition, November 2003.

[23] G. S. Stiles, V. Raju, and L. Rong. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 63–81, 2003.