Experience with safe dynamic reconfigurations in component-based embedded systems

Juraj Polakovic[1,2] Sébastien Mazaré[1] Jean-Bernard Stefani[2] Pierre-Charles David[3*]

[1] FranceTelecom R&D, MAPS/AMS Lab, Grenoble, France
{juraj.polakovic,sebastien.mazare}@orange-ftgroup.com

[2] SARDES Project, INRIA Rhône Alpes, Grenoble, France
jean-bernard.stefani@inria.fr

[3] OBASCO Group, EMN/INRIA, Lina, Nantes, France
pierre-charles.david@emn.fr

Abstract

Supporting dynamic reconfiguration is required even in highly constrained embedded systems, to allow patches and updates to the embedded systems software, and to allow adaptations to changes in environmental and operating conditions without service interruption. Dynamic reconfiguration, however, is a complex and error prone process. In this paper we report our experience in implementing safe dynamic reconfigurations in embedded devices with limited resources. Our approach relies on a component-based framework for building minimal and reconfigurable operating systems and the use of a domain specific language (DSL) for reconfiguration.

1 Introduction

Dynamic reconfiguration refers to the process of modifying a system's structure and behavior during its execution. Even in memory and energy constrained devices such as networked sensors or embedded appliances, dynamic reconfiguration is required to allow the application of software patches and security updates, the application of functional updates (e.g. introducing a new protocol), or the adaptation to new environment and operating conditions, while ensuring a continuity of service.

Dynamic reconfiguration, however, is a complex process, which can be very error-prone, as the reconfiguration programmer must maintain both the architectural and behavioral integrity of the system under modification. For example when replacing a module in a multi-threaded operating system, a quiescent state must be achieved, and the interface offered by the module must match what the rest of the system expects. (for a discussion of the intricacies of dynamic reconfiguration see e.g. [4, 33]).

Existing software infrastructures for embedded devices with limited resources such as e.g. TinyOS [20], Contiki [9], SOS [17], Mantis [3], FlexCup [25], either do not support dynamic reconfiguration (TinyOS), or provide low-level mechanisms for dynamic reconfiguration — typically dynamic linking of modules or components — that do not alleviate the issues involved in programming safe reconfigurations.

This paper reports our experiences with the implementation of a support for safe dynamic reconfigurations on embedded devices with limited resources. We rely on the THINK framework for the construction of component-based operating systems [12], that implements the FRACTAL reflective component model [5], to construct reconfigurable embedded software [27]. Safety of reconfigurations is achieved by a combination of architecture-based component replacement, and the use of a novel domain specific language (DSL), called FScript [7], that allows the succinct expression of modifications to a running architecture, and that

*This work has been done while the author was a post-doctoral fellow in France Telecom R&D.

supports various sanity checks on reconfiguration programs. This paper discusses our approach, and the associated support for the FScript language, with respect to goals of flexibility, efficiency, safety and simplicity of safe reconfigurations.

The remainder of this paper is structured as follows. In section 2 we describe our use case and discusses the challenges faced when designing and implementing a operating systems supporting safe reconfigurations. Section 3 details our approach for constructing reconfigurable operating system kernels and programming reconfigurations. The support for the FScript language is crucial, we show in section 4 the obtained implementation results. Section 5 discusses the lessons we learned and the future work. Follows a discussion of existing research and section 7 concludes the paper.

2 Use case and Challenges

2.1 Use case: the Cognichip

Our final objective is to implement safe reconfigurations on our target platform, called the Cognichip, based on the Atmel AVR ATmega128 8-bit micro-controller with $4kb$ data memory and $128kb$ code memory. We are currently experimenting approaches to construct mobile intelligent networked objects with cognitive radio [26] on top of this platform.

At run-time, the Cognichip can be extended with different plugin devices, such as various sensors or device controllers enabling the Cognichip to act as a small intelligent information router. This feature requires to add additionnal drivers at run-time. Furthermore, informations from these devices are handled by applications installed either by the user or by a third-party, like the house internet provider, that also can be changed during the execution.

With the THINK framework we already construct applicative operating systems tailored to this platform [21]. The operating system and the application are built as a unique THINK-based component-system. We want to enable dynamic reconfiguration of the system and application components, in order to support bugfixes, reconfigure the underlying radio protocol components and add or remove device drivers. The addition of device drivers may require the reconfiguration of some low-level system components, for example adding a new brick in the protocol stack that requires a different timer implementation.

2.2 Challenges

Designing and implementing dynamic reconfiguration mechanisms in an embedded operating system is a challenging task. Issues related to reconfiguration mechanisms have an impact on the design of the system itself and must be considered early in the development phases.

In order to replace a component in a system, first, the part of the system to be reconfigured (for brevity we call it the reconfiguration target) must be clearly identified. Then, before the reconfiguration takes place, the reconfiguration target must reach a stable state. A common notion of stable state is that of quiescent state, i.e. a state in which no activity currently takes place in the reconfiguration target. When this quiescent state is detected, the state of the reconfiguration target must be captured and transfered to the new component and the change of configuration can now take place. The change of configuration can imply changing some attributes, modifying the connections between modules, as well as altering the software architecture of the reconfiguration target. Before resuming the execution after the configuration change, the references to the old component must be redirected to the new one.

As such, reconfiguration mechanisms in embedded operating systems raise new challenges beyond the operating system construction. We consider four main goals for a dynamic reconfiguration mechanism, already described by Hicks and Nettles [19], however adapted to embedded operating system construction and to the above described use-case:

Flexibility Any part of the system should be reconfigurable, especially the system should not impose any non-reconfigurable core set of components.

Efficiency and Minimality The reconfigurable operating system support for safe reconfigurations must respect performance constraints of the target platform. For embedded devices these limitations can vary from memory usage or CPU consumption, to power consumption or timing requirements of reconfigurations.

Safety By *safety* of reconfigurations we mean that erroneous reconfigurations won't compromise the system consistency, i.e. the resulting architecture is valid. The reconfigurable system must also guarantee a behavior correctness during reconfigurations, e.g. that a module is not removed while accessed.

Simplicity The reconfiguration process, including programming complex reconfigurations, must be simple for the reconfiguration programmer, in order to minimize the introduction of errors.

3 Foundations: extensive use of a component model

3.1 FRACTAL and THINK

The FRACTAL component model FRACTAL is a hierarchical and reflective component model intended to implement, deploy and manage a wide range of software systems including operating systems and middleware [5, 6].

A FRACTAL component is both a design and a runtime entity that constitutes a unit of encapsulation, composition and configuration. Components provide *server interfaces* which are the access points to the services that they implement. They express their service requirements via *client interfaces*. FRACTAL distinguishes two kinds of components. *Primitive components* are implemented in a host programming language (e.g. C, Java) and can be seen as black boxes providing and requiring services through their interfaces. *Composite components* correspond to a composition of other components (called subcomponents), either primitives or composites. The existence of composite components makes the FRACTAL Component Model a hierarchical component model.

Components in the FRACTAL component model interact via client/server *bindings*. A binding constitutes a communication path between components and can implement arbitrary forms of communication (e.g asynchronous requests, synchronous requests/replies, multicasting and so forth). The simplest form of binding is a language reference (e.g. a method invocation).

A FRACTAL component logically comprises two different parts. The internal part, that we call *content* implements the functional interfaces of the component. The content is encapsulated by a *membrane* which can implement control over its behavior. In addition to the functional interfaces of a component, the *membrane* can provide an arbitrary set of control interfaces.

The presence of control interfaces makes FRACTAL a reflective component model where control interfaces provide the means to observe and manipulate the internal structure of a component. An important point is that the FRACTAL component model does not mandate a fixed and predefined meta-object protocol (i.e. a set of control interfaces). Instead, a programmer can define and implement his own set of control interfaces.

FRACTAL defines some standard control interfaces in order to manipulate a component's interfaces, its subcomponents, its client bindings, attributes, or its lifecycle (respectively called `ComponentIdentity`[1], `ContentController`, `BindingController`, `LifeCycleController`).

Architecture Description Language FRACTAL components and FRACTAL component architectures can be described using an architecture description language (ADL). The FRACTAL ADL is a high-level declarative language in which programmers express software configurations in terms of interfaces, attributes, component compositions and bindings. Using the `membrane` construct, the programmer can specify which membrane to apply to a given component. An example of an ADL description is given in the figure 1.

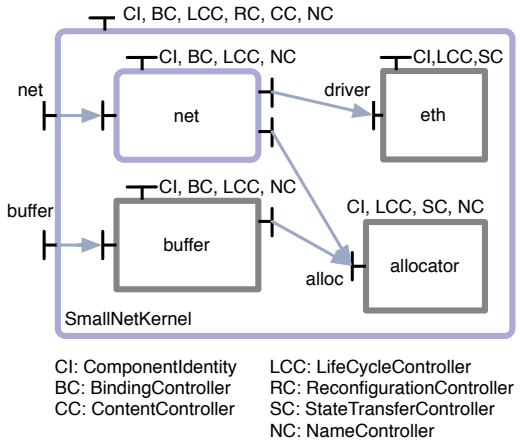
THINK THINK is a general framework for building component-based systems and especially operating system kernels. The framework comprises a C implementation of the FRACTAL component model, an ADL with the associated ADL-to-C compiler, we call THINK ADL compiler, and a component library, called Kortex.

The initial version of THINK [12] was based on a flat component model. Our current version is now entirely based on the hierarchical FRACTAL component model. Thanks to the reflective capabilities of FRACTAL, a kernel architecture can be retrieved at run-time. If the kernel has a static architecture, i.e. it doesn't contain any component factories that could alter its architecture, the run-time kernel architecture is equivalent to the initial ADL description.

In a THINK-based OS everything is a component and with an adapted reconfiguration base mechanism, every component is reconfigurable. There is no predefined non-reconfigurable core, nor a core set of required components. In this way, we can build customized reconfigurable operating systems, including only the needed components. Such systems can range from small devices like networked sensors [16] to bigger devices, like handhelds, offering more functionality.

THINK-based OS are built using the THINK ADL compiler. This compiler translates the ADL code to ANSI C and compiles and links the generated code with

[1] called simply `Component` in the latest FRACTAL specification




```
composite SmallNetKernel {
  provides net.api.Net as net
  provides util.api.Buffer as buffer

  contains net = net.lib.tcpip
  contains eth = chip.net.tulip
  contains buffer = util.lib.buffer
  contains alloc = memory.lib.malloc

  binds net.driver to eth.driver
  binds net.alloc to alloc.alloc
  binds buffer.alloc to alloc.alloc
  binds this.net to net.net
  binds this.buffer to buffer.buffer

  membrane Reconfig_ThreadCounting
}
```

Figure 1: A simplified view of a simple reconfigurable THINK-based kernel and its ADL description. The kernel provides a network interface and a buffer interface. The `eth` component implementing the network driver is architecture-dependent. The FRACTAL control interfaces are added to the components by the THINK ADL compiler based on the `membrane` keyword.

component implementations (also written in C) using a standard C compiler and linker. The generated C code obeys to a binary format for components detailed in [12].

We have developed a THINK-based component library, called Kortex, including various standard operating system functions and architecture-dependent components. For example the library provides several implementations of schedulers, allocators, file systems, ELF linker and loader or network communication protocols (TCP/IP, bluetooth, GPRS..), as well as low-level components on several platforms ranging from ARM or AVR cores to PowerPC architectures (interrupt handlers, network drivers or display drivers...).

3.2 Dynamic reconfiguration in THINK-based OS

The THINK framework was lately enhanced with a number of base mechanisms for replacing safely a component without interrupting the service [27]. These mechanisms provide the detection of quiescent state, state transfer and reference redirection and are implemented as FRACTAL control interfaces.

The THINK framework provides several implementations for these interfaces. For a given component, the kernel developper chooses the appropriate set via the ADL `membrane` keyword and the THINK ADL compiler includes the appropriate interface implementations for reconfigurable components. Note this approach is extremely flexible and allows to specify different mechanisms at the scope of a single component.

- *Quiescent state* Quiescent state algorithm is provided via the component's `ReconfigurationController` interface implementation. In our previous work [27], we implemented two quiescent state algorithms for THINK-based kernels – *thread-counting* and *dynamic interceptors*.
- *State transfer* We have defined a `StateTransferController` interface that gives access to a component's internal state and allows us to build complex state transfers. It is up to the component programmer to implement this interface.
- *Reference redirection* Once a reconfiguration succeeded, client references are redirected using the standard `BindingController` interface of the client components.
- The actual architecture modifications are performed by the `ContentController` interface implementation that can add or remove components from a composite component.
- The `LifeCycleController` interface serves to initialize (start) or to stop a component, used to initialize a hardware driver or shutdown properly a device.

Consider the example of the `SmallNetKernel` shown in figure 1. The kernel developer writes the ADL description for the `SmallNetKernel` as shown in the figure below the graphical view of the kernel. Based on the membrane specification of this ADL description, the THINK ADL compiler will automatically generate FRACTAL control interface implementations for the `SmallNetKernel` component, concretely including the thread-counting quiescence algorithm. The graphical view shows the resulting set of control interfaces of the components.

3.3 The FScript DSL

The FRACTAL component model is defined in terms of APIs which make it possible both to *discover* the structure of a FRACTAL application and to *reconfigure* it at runtime. However, programming dynamic reconfigurations directly at this level make the code verbose, difficult to understand, and prevent analyzes to guarantee the behaviour before of the reconfigurations executing them, which can be essential for critical systems. FScript [7] is a Domain Specific Language [34] designed to overcome these limitations while retaining FRACTAL's advantages. FScript can be used to navigate intuitively inside a FRACTAL architecture and select parts of it, on which reconfigurations (either primitive FRACTAL operations or user-defined scripts) can then be applied.

Here is an example of a simple reconfiguration script programmed in FScript which illustrates all of FScript constructs.

```
action auto-bind(c) = {
 // Selects the interfaces to connect
 clts := $c/interface::*[required(.)]
                        [not(bound(.))];
 foreach i in $clts do {
   // Search for candidates
   // compatible interfaces
   srvs := $c/sibling::*/interface
                  ::*[compatible($i, .)];
   if (not(empty($srvs))) {
     // Connect one of these candidates
     bind($i, one-of($srvs));
   }
 }
 return $c/interface::*[required(.)]
                       [not(bound(.))];
}
```

This defines a new reconfiguration action named `auto-bind`, which automatically connects a component's required interfaces by discovering the compatible server interfaces on sibling components. The body of the action consists in a sequence of simple statements (assignments, procedure calls and return) and control structures (iteration and conditionals).

As can be seen in this example, FScript uses a special notation to navigate inside the architecture and select elements from it. This notation, called FPath, is roughly inspired by XPath [36] and supports the same kinds of queries on FRACTAL architectures that XPath supports for XML documents. FPath expressions are used to navigate inside the architecture to select specific elements. The architecture is seen as a directed graph where nodes represent components, their interfaces, methods and attributes. Theses nodes are connected by labeled arcs representing the relationships between them: for example, a component node is connected to all its interfaces' nodes by arcs labeled `interface`. Each step of a path starts from an initial set of nodes and selects a new set by following one of these relationships. The resulting nodeset can then be filtered by name or by complex predicate expressions (including embedded path expressions).

As a concrete example, the second FPath expression in the above example (line 6) can be read as: "Starting from component `$c`, first select all its siblings (components which share at least one direct parent) in the architecture, whatever their name is (`/sibling::*`). Then, given these new components, select all their interfaces (`/interface::*`), but return only those for which the predicate in brackets holds. In this case, the predicate tests whether its parameter (denoted by a dot `.`) is compatible with the interface in the iteration variable `$i`." In short, this request will select all the interfaces owned by siblings of the parameter component which can be bound to `$i` (itself a required interface of `$c` which is not bound yet, see line 3).

FScript provides an extensible library of primitive functions and actions which gives the user access to all the features of the FRACTAL API. These primitives can be combined to create complex reconfiguration scripts using a voluntarily limited set of the control structures which, with the interdiction of recursive definitions, enable us to guarantee the termination of all reconfigurations: classical *conditionals* (`if/then/else`), *iteration* (`foreach i in path do { body }` on the result of an FPath expression (`path`), which always return a finite set of elements, and finally explicit and early *return* from an action (`return`).

Compared to the use of the standard FRACTAL APIs in a general purpose language, FScript offers better syntactic support for navigation and more dynamicity. In addition, the FScript implementation guarantees the consistency on the reconfigurations (termination, atomicity, validity of the resulting configuration, see [7]). A reconfiguration which would take several pages of verbose and error-prone C or Java code using the standard APIs directly can often be expressed in a few lines of FScript code, which is both much more readable and easier to check for validity.

4 Results

This section discusses our results obtained when implementing reconfigurable operating systems with the THINK framework and writing reconfigurations with the FSCRIPT DSL.

4.1 Using the FScript language

In THINK-based OS reconfigurations are performed in terms of operations allowed by the FRACTAL component model. The following sequence is necessary to replace the `alloc` component in the `SmallNetKernel` kernel shown in figure 1. The new component is called `new_alloc`.

1. add the new component to the `SmallNetKernel`
2. suspend the execution of the `alloc` component
3. stop the `alloc` component (for example driver shutdown)
4. (retrieve state from the `alloc` component)[2]
5. (inject state to the `new_alloc` component)[2]
6. redirect the `net` component to use `new_alloc`
7. redirect the `buffer` component to use `new_alloc`
8. start the `new_alloc` component (for ex. initializations)
9. resume the execution in the `SmallNetKernel` component
10. remove the unused `alloc` component

[2]For the sake of simplicity, we wont consider the state transfer in this prototype evaluation. However, complex state transfer mechanisms can be built using the StateTransferController controller to retrieve or set component state during reconfiguration.

In FScript we would write the following reconfiguration code to achieve this reconfiguration. Note we annotated with comments the FScript code with the corresponding operations listed above.

```
k = $root/child::kernel
n := new('new_alloc');
o := $root/child::alloc

add($k, $n);     // 1
suspend($k, $k/child::alloc);  // 2
stop($o);        // 3
unbind($k/child::*/interface::alloc);
bind($k/child::*/interface::alloc,
     $n/interface::alloc);     // 6,7
start($n);       // 8
resume($k);      // 9
remove($o);      // 10
```

Expressiveness gain The above shown FScript code shows the simplicity of our approach. In the shown code, the programmer expresses all needed operations to perform a dynamic reconfiguration. An equivalent C code is much harder to understand. As an example, the `alloc` component provides an `alloc` interface. The re-binding of all client components (operation 6,7) to use this interface is written in 10-20 lines of C code, whereas it holds in two lines in FScript - first we unbind all client interfaces, then we bind them. The equivalent C code we considered was without error handling that adds to the complexity and decrease readability of the C code.

Correctness checks Correctness is obtained by simulation of the reconfiguration at compile-time and by generating error handling code, thus at run-time.

At compile-time, the FScript compiler checks if the code is conforming to the target architecture. It verifies the conformance of interface types, the existence of components and the correct use of suspend/resume quiescent state operations. Performing these verifications at compile-time reduces the resulting size of the compiled reconfiguration code to be loaded by the target device in its constrained environment.

An explicit error-handling code can be written by the reconfiguration programmer in the FScript program, serving to perform additional architecture verifications at run-time. A checking code is automatically generated by the FScript compiler. It serves to make low-level C checks, as null pointers etc. preventing the possible crash of the system.

Kernel architecture evolutions FScript reconfiguration programs are easier to adapt and maintain over time. This is not the case of hand-written C code for reconfiguration.

Consider the following scenario: the above described reconfiguration is commonly applied to change the allocator performances in some of the deployed embedded devices running a first generation of the `SmallNetKernel`, without the `buffer` interface. For some reasons the `SmallNetKernel` was updated and the new version provides a `buffer` implementation that uses the allocator component, as shown in figure 1. We need to change only one line in the FScript program to fit this scenario - the bind operation.

4.2 Implementing support for FScript

We found that there were several possible strategies for implementing the FScript compiler, the central piece of our approach. We first discuss the implementation strategies and show our implementation together with the obtained results in the following section 4.3.

Native FScript interpreter The first general idea is to provide an interpreter in the native environment, embedded in the target operating system. All FScript programs are addressed to this interpreter and are interpreted and executed on the target platform. This implementation strategy is shown in the figure 2.

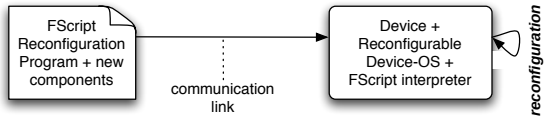


Figure 2: Native FScript interpreter.

Offline compiler The second strategy consists in compiling FScript programs on a remote host into a binary form, ready to be loaded and executed on the target platform. This approach alleviates the requirements on the reconfiguration support on the target platform. All necessary program verifications can be performed at compile time, resulting in a safe reconfiguration code. However, in such a distributed architecture, the FScript compiler, running on the reconfiguration host, has to know or retrieve the software architecture of the target OS and probably some additional meta-data. The retrieval of the software architecture is enabled with the use of the FRACTAL component model where component architecture representation is explicitly maintained at run-time.

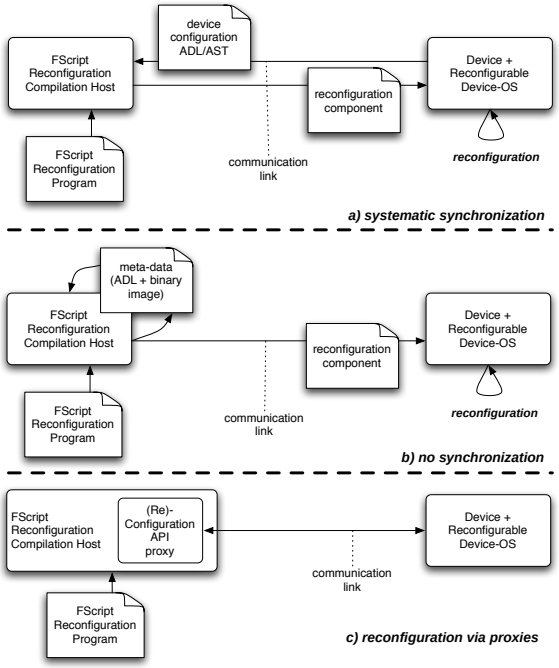


Figure 3: Offline compiler: different implementations of synchronization between the FScript compiler and the target OS.

There are several variations of this approach, shown in figure 3. The first variant (we call *systematic synchronization* in figure 3a), retrieves systematically the current configuration of the device's OS on each reconfiguration request. The variant shown in figure 3b, *no synchronization* makes assumption that the software architecture of the target subsystem to reconfigure evolves only through reconfigurations. For the sake of completeness, figure 3c, *reconfiguration via proxies*, shows an implementation of the synchronization using a proxy component system on the reconfiguration host having identical configuration as the target system and that forwards all reconfiguration actions to the former.

The choice of a solution for the FScript support implementation will depend on the hardware constraints of the target platform, criteria defined by the operating system or by the network provider the embedded system is connected to. We implemented an offline FScript compiler without synchronization and show the obtained implementation results in the next section.

4.3 An Offline FScript Compiler

For evaluation purpose we implemented an FScript compiler without synchronization (Figure 3b). In order to validate the minimality of our approach and evaluate the suitability for a platform like the above described Cognichip, we built a small prototype kernel on an 32-bit ARM-based handheld device. We are interested in evaluating the memory overhead of the run-time support for FScript and the amount of data transferred to the reconfigurable device.

Implementation of the FScript compiler Our FScript compiler without synchronization works on the ADL description used to instantiate the target system, this is sufficient, as FScript semantics are defined exclusively in terms of FRACTAL architecture elements captured by the ADL. The FScript compiler generates the C implementation of a reconfiguration component.

Along with the new instances, the reconfiguration component is compiled as a relocatable THINK-based component with our THINK ADL compiler to produce an ELF[3] file ready to be sent to the target platform.

The target operating system provides a minimal run-time for executing reconfiguration components:

- *FScript helper functions* In order to minimize the transferred reconfiguration C code, some often used functions are provided by the run-time.
- *loader and linker* The linker functionalities are restrained to be capable of only loading the reconfiguration ELF file. It performs some symbol relocation and resolves references to the FScript helper functions, together with some standard functions (symbols), like `printf`.
- *introspection component* In the generated C code, the real component locations are not resolved. The resolution is done at run-time by the introspection component, using the introspection capabilities of FRACTAL architectures already available in the operating system.

Implementation results Table 1 shows the sizes of the necessary components to build a run-time support for the binary reconfiguration code compiled from FScript. We separate the code size (`text` section) from the data size (`data` and `bss` section). On architecture like the AVR, the code is copied into the flash memory, whereas the data into the main memory. The data comprises C compiler allocated structures, like global variables, especially it doesn't give any information on stack or heap usage. As already stated, the linker has only minimal functionalities to satisfy the linking of the reconfiguration component.

	code size	**data size**
Introspection	$2.7kb$	$100b$
Loader and linker	$4.9kb$	$300b$
FScript helper	$1.3kb$	$0b$
Total run-time support	$8.9kb$	$400b$

Table 1: Sizes of the different elements of the FScript execution run-time.

In the table 2 we evaluated the size of the ELF file of the reconfiguration component. The functional content of the ELF file, i.e. reconfiguration code and new components, is about half of the file size. In the ELF header we find a symbol table, a string table and two relocation tables (code and data sections). The space occupied by the relocation tables are minimized due to the inlined nature of the THINK component structures we generated, the symbol and string tables occupy almost the rest of the header. The sizes of these table-entries are directly related to the C symbol names (variables, functions etc.)

The size of the ELF header is directly related to the structure of the C code of the new components and of the generated C structures for all THINK-based components. The size is proportional to the amount of components in the file

Reconfiguration code	$1.5kb$
The new component (`sbrk`)	$0.5kb$
THINK component structures	$0.2kb$
Total content:	$2.3kb$
ELF header	$2kb$
strings in the ELF tables	$\sim 1kb$
Total size of the ELF-file	$4.3kb$

Table 2: Details of the reconfiguration ELF file for the allocator reconfiguration example.

Table 3 summarizes the characteristics of the offline compilation approach and compares it to what we could expect from a native interpreter (we didn't implement). We are interested in comparing the amounts of data

[3]The use of the ELF format is motivated by the availability of associated code and tools.

transferred to the reconfigurable device and the impact on the memory usage of the FScript support as it determines the suitability of the approach for constrained devices. The shown sizes are relative to the above described reconfiguration of the `alloc` component. We postpone the discussion of these results to the section 5.1.

	offline compiler	**native interpreter**
Transf. data type*	ELF	FScript
Transf. data size*	$4.5kb$	$270b$
FScript run-time	$8.9kb/0.4kb$	n.a.

Table 3: Summary of the results of our FScript offline compiler, compared to what we could expect from a native interpreter. *) the shown sizes are relative to the reconfiguration of the `alloc` component.

Table 4 shows the size of the reconfigurable THINK-based kernel served for this evaluation. All kernel components are reconfigurable, the kernel contains an FScript support (helper functions, loader and linker) and a simple communication stack over the radio line. In the data sections (`data` and `bss`) we find static component data, buffers and component meta-data structures generated by the THINK ADL compiler (about $1kb$).

	code	**data**
a reconfigurable kernel	$109kb$	$13kb$

Table 4: Code and data sizes of a reconfigurable THINK-based kernel with an FScript support and a radio communication link.

4.4 Other considerations

All variants of the FScript support, either interpreted or compiled, will use the reconfiguration API provided by the reconfigurable component's FRACTAL control interfaces, as described above in section **??**. The implementation of these interfaces is independent of the actual support for the FScript language.

In previous work [27], we evaluated the two provided implementations for detecting quiescent state: thread-counting and dynamic interceptors. Each implementation satisfies different criteria, like CPU usage, memory consumption or assumptions about the execution model. A more deep discussion and evaluation can be found in the cited paper.

5 Discussion

5.1 Lessons learned and Limitations

We found that our approach for providing safe reconfigurations in an operating system fulfills the goals of flexibility, safety and simplicity. However we didn't yet achieve the goals of efficiency and minimality, in order to build an FScript execution support on the Cognichip target platform.

Flexibility Using the FRACTAL hierarchical component model allows reconfigurations to be properly scoped and to take place at different levels of granularity. Changing a single primitive component or a whole subsystem is done in the same way. Using this approach we were able to replace a single component, like the allocator, and the whole application.

Safety Our Safety of reconfigurations is achieved by using the FScript language. FScript programs are subject to compile time checks that help the reconfiguration programmer avoid non-trivial errors. The FScript compiler is responsible for analyzing if the reconfiguration performed results in a valid architecture and it also generates run-time error-handling code.

Simplicity Our approach is simple thanks to the use of FRACTAL-based technologies at every stage of the reconfiguration process. We build reconfigurable operating systems, using a FRACTAL implementation, we program reconfigurations in a language based exclusively on elements of a FRACTAL architecture.

Efficiency and Minimality The actual sizes of a reconfigurable kernel with an FScript support for an AVR platform may vary from those reported in the previous section (due to different code compactness and different data sizes). The above results show however a lack of the implemented approach – it requires extra memory usage on the target platform, inadapted for platforms like the Cognichip, with $4kb$ or $8kb$ main memory.

Indeed, the size of the ELF header is proportional to the number of components (due to the nature of the

generate meta-data structures for THINK-based components), the overall size of the transferred file depends on the number of new components and the reconfiguration code is proportional to the complexity of reconfigurations. Thus, the processing of the received ELF file requires extra memory usage.

5.2 Perspectives

The above discussed prototype could be improved in several ways.

Optimize generated component structures The ELF-header overhead is partially due to the structure of the C code generated by the THINK ADL compiler. For a component the compiler generates several cross-referenced global symbols. Each reference of such a symbol requires an entry in the relocation table found in the ELF header. We are currently experimenting the generation of inlined component structures revoking the necessity of cross-referenced symbols. This is a more general work on optimizations of component-based system architectures.

Loader Our prototype uses a simple loader working on in-RAM copies of the ELF file. On architectures like the AVR, where the RAM size is limited, we plan to build a more efficient loader working with the flash memory as temporary storage.

External configuration representation and meta-data The described implementation of the FScript interpreter works with an ADL representation of the device's configuration. The interpreter generates a code where real component locations are resolved at run-time. We envisage to include binary meta-data, as effective component locations, to the configuration representation gathered by the target device prior to reconfiguration. This would probably lead us to define an alternative binary format for the transmission of the reconfiguration component.

Offline pre-linking We are currently working on a prototype implementation of the FScript compiler without the necessity of a run-time linker. All linking is done on the compilation host, linking locations are predetermined before the binary reconfiguration component is sent to the target. The memory usage of this prototype is thus minimal, this approach has severe limitations in flexibility.

FScript syntax evolution Our actual FScript compiler checks the conformity of the usage of the `suspend` and `resume` operations for the quiescent state. This check is complex and we still found cases difficult to detect. Blocks in the FScript language would be useful to provide syntactically scoping rules to this quiescent state condition. A programmer would write

```
on suspended(myComponent) {
  ...
  //reconfig-actions
}.
```

The FScript compiler could then just insert the `suspend` and `resume` operations.

Error handling The reconfiguration code as it is generated by the FScript compiler contains little error handling assuming that all verifications were performed by the interpreter. We envisage to make this code more robust and evaluate the impact on its size. A C++ exception mechanism implementation appears to be a promising way to achieve a proper error handling.

6 Related Work

Dynamic reconfiguration has been heavily explored in research areas ranging from programming languages, down to middleware and operating system kernels. In the following, we restrict the analysis to reconfigurable operating systems. We organize our discussion around two research areas. First we discuss reconfigurable operating systems with respect to the reconfiguration mechanisms and the offered safety mechanisms. And second we discuss operating systems for constrained embedded devices, like networked sensors, that with exception of Mate [23], do not offer safe reconfigurations.

Reconfigurable operating systems General-purpose operating systems, such as Linux or Windows provide limited support for dynamic reconfiguration, typically limited to certain functionalities, like device drivers. It is possible to load and unload kernel modules, but for example replacement of a module while in use by the system is not possible. All such reconfigurations are performed manually and a verification mechanism, as implemented in this paper, is not available.

Component-based OS, such as OSkit [13] or eCos [10], provide a way to build customized and minimal kernels, based on compile-time selection of components to be included into the kernel. These systems also provide an architecture description language (ADL), such as Knit [29], to assist the assembly of the kernel. However, these systems have no support for dynamic reconfiguration.

Compared to monolithic operating systems, micro-kernels (L4 [24], Chorus [30], Kea [35] or Pebble [14]) are a step further in providing reconfigurability - a user-level server is the unit of reconfiguration. However, a micro-kernel itself is not reconfigurable and if reconfigurability is implemented at the level of user-level servers, the system pays the price of the time-consuming IPC communication between these servers. An Exokernel approach [11] allows kernel developers to build systems on top of minimum hardware abstraction, however it is also up to the kernel developer to implement a reconfiguration support.

SPIN [2], provides a safe way to extend the kernel, by writing extensions as *spindles*. Compared to our approach, SPIN has several limitations. First, an underlying kernel core itself is not reconfigurable. Second, extensions are only limited to some predefined parts of the kernel. Third, the interactions between extensions and the kernel are expensive - an extension is a handler reacting to an event raised by a kernel module. And finally the safe extension mechanism requires a run-time compiler and verifier, which makes this approach unsuitable for constrained embedded devices.

In VINO [31, 32], all reconfigurations are handled as transactions, largely using locking to synchronize the access to kernel modules. As such, transactions offer a basis for implementing a safe reconfiguration mechanism. The overhead of the transaction mechanism makes it inadapted to be used in embedded devices with limited resources. Reconfigurable operating systems including Synthetix [28], MMLite [18] or more recently K42 [1, 33] provide mechanisms for dynamic reconfiguration at a fine grain. Synthetix and MMLite use read-write locks to synchronize accesses to a reconfigurable component. K42 supports reconfiguration through a mechanism, which consists in introducing interceptors at run-time, resulting in no run-time overhead. Common to these systems is the fact that reconfigurations are still hand-written and as such do not alleviate the issues involved in programming safe reconfiguration.

Operating systems for networked sensors TinyOS [20] is one of the first operating systems targeting networked sensors. TinyOS is implemented using the nesC programming language [15]. TinyOS is not reconfigurable, however, several different approaches exist on top of TinyOS in order to provide application reconfigurability. For instance XNP [22] allows to download and reinstall a new system image. XNP requires a reboot of the system and as such doesn't provide any service guarantee during the reconfiguration.

FlexCup [25] is another mechanism built on top of TinyOS that provides a dynamic reconfiguration mechanism, allowing to reconfigure applications at the granularity of a TinyOS component. The mechanism relies on meta-data, generated during the compilation of the system, and a run-time linking mechanism based on these meta-data. However with FlexCup, TinyOS applications must be segmented into arbitrary binary components that can be reconfigured subsequently. Using the THINK approach for reconfigurations, all components found in a kernel are reconfigurable. Also, in a THINK-based operating systems, components are run-time entities and all necessary meta-data for retrieving and relinking a component are available through the use of the FRACTAL reflective component model.

Contiki [9, 8] is a reconfigurable modular operating system for networked sensors. A Contiki system uses a flat module architecture, a module is the unit of reconfiguration. Thus Contiki offers only a fixed granularity of reconfigurations, whereas in a THINK-based system, because of the hierarchy offered by the FRACTAL component model, reconfigurations can involve the whole application implemented by a complex component, or only its subpart implemented in some primitive component. Technically the approach shown in this paper is similar to Contiki [8], however, using a THINK-based OS with an explicit architecture representation, provides us with a support to verify the reconfigurations before execution. Contiki defines a non-reconfigurable core, in THINK-based OS, everything is a component, thus with an appropriate method, everything is reconfigurable. SOS [17] achieves reconfiguration using loadable modules. The SOS approach for reconfiguration is similar to Contiki.

Mantis [3] offers a remote shell for requesting reconfigurations. A new system image must be loaded into the ROM and the system is restarted without any service guarantee during the reconfiguration.

Maté [23] is a virtual machine (VM) built on top of TinyOS. Maté applications are written with a limited

set of virtual machines instructions. Reconfigurations are performed as replacements of such applications running on top of the virtual machine. Compared to our approach, Maté achieves different trade-off between the four goals of a safe dynamic reconfiguration mechanism. The granularity of reconfigurations in Maté is coarse (at the applications level), the core of the system, the virtual machine itself, can't be reconfigured, thus our approach using THINK is more flexible. With the byte-code interepretation, the VM approach achieves good safety of reconfigurations. The radio communication in networked sensors being the most expensive resource, virtual machines provide energy-efficient update mechanisms, but VMs are more energy expensive during normal system execution. With respect to the energy efficiency we achieve a different trade-off with the THINK approach.

7 Conclusion

In this paper we discussed an approach for constructing safe reconfigurations of operating systems by using a reconfiguration DSL, called FScript. Based on a use-case for intelligent networked objects, we considered four goals for a mechanisms for safe reconfigurations – flexibility, efficiency and minimality, safety and simplicity. Our implementation of the FScript offline compiler for an ARM-based platform fulfills the above goals, however it reveals a prohibitive usage of the memory on the target platform, mainly due to the transfer format of the loadable code, that we will address in our future work by offline pre-linking the loadable code.

The THINK framework is freely available at `http://think.objectweb.org`.

References

[1] A. Baumann, G. Heiser, J. Appavoo, D. DaSilva, O. Krieger, R. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference*, April 2005.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[3] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *MONET*, 10(4), 2005.

[4] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and Practice. *IEE Software Engineering Journal*, 8(2), 1993.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11–12), 2006.

[6] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model. http://fractal.objectweb.org.

[7] P.-C. David and T. Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, July 2006.

[8] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Runtime dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, 2006.

[9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, 2004.

[10] eCos. http://sources.redhat.com/ecos.

[11] D. Engler, M. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[12] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think: a software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX, June 2002.

[13] B. Ford, J. Lepreau, S. Clawson, K. V. Maren, B. Robinson, and J. Turner. The Flux OS Toolkit: Reusable Components for OS Implementation. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[14] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *Proceedings of the USENIX Annual Technical Conference*, June, 1999.

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, 2003.

[16] F. Germain, M. Ghozzi, J.-P. Laval, T. Jarboui, and F. Marx. The Cognichip: a flexible, lightweight spectrum monitor. In *COGnitive systems with Interactive Sensors (COGIS)*, 2006.

[17] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

[18] J. Helander and A. Forin. MMLite: a highly componentized system architecture. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, 1998.

[19] M. W. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6), 2005.

[20] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2000.

[21] T. Jarboui, F. Marx, J.-P. Laval, M. Ghozzi, F. Germain, and O. Lobry. The Cognichip: A flexible, lightweight spectrum monitor. In *COGnitive systems with Interactive Sensors (COGIS)*, 2006.

[22] J. Jeong, S. Kim, and A. Broad. Network reprogramming. TinyOS documentation. 2003. http://www.tinyos.net/tinyos-1.x/doc/ NetworkReprogramming.pdf.

[23] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[24] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[25] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *European Workshop on Wireless Sensor Networks*, 2006.

[26] J. Mitola. *Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio*. PhD thesis, Royal Institute of Technology (KTH), May 2000.

[27] J. Polakovic, A. E. Özcan, and J.-B. Stefani. Building reconfigurable component-based OS with THINK. In *32nd Euromicro Conference on Software Engeneering and Advanced Applications (SEAA), Track on Component-Based Software Engeneering*, 2006.

[28] C. Pu, T. Autrey, A. P. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[29] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2000.

[30] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *In Computing Systems*, Vol. 1(4), 1988.

[31] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 1996.

[32] C. Small and M. Seltzer. Structuring the kernel as a toolkit of extensible, reusable components. In *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, Aug. 1995.

[33] C. Soules, J. Appavoo, K. Hui, R. Wisniewski, D. D. Silva, G. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June, 2003.

[34] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6), 2000.

[35] A. Veitch and N. Hutchinson. Dynamic Service Reconfiguration and Migration in the Kea Kernel. In *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS)*, 1998.

[36] World Wide Web Consortium. XML path language (xpath) version 1.0. W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xpath`.