Building Modular Real-time software from Unified Component Model

1,2 K.A. Mallachiev <mallachiev@ispras.ru>
1,2,3,4 A.V. Khoroshilov <khoroshilov@ispras.ru>
[1] Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[3] Moscow Institute of Physics and Technology,
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] Higher School of Economics.
20, Myasnitskaya Ulitsa, Moscow 101000, Russia

Abstract. Modern real-time operating systems are complex embedded product made by many vendors: OS vendor, board support package vendor, device driver developers, etc. These operating systems are designed to run on different hardware; the hardware often has limited memory. Embedded OS contains many features and drivers to support different hardware. Most of the drivers are not needed for correct OS execution on a specific board. OS is statically configured to select drivers and features for each board. Modularity of OS simplifies both configuration and development. Splitting OS to isolated modules with well-specified interfaces reduces developers' needs to interact during joint development. The configurator, in turn, can easily compose isolated components without component developers. We use formal models to specify components and their composition. Formal model describes the behavior of components and their interaction. Usage of formal models has many benefits. Models contain enough information to generate source code in C language. Our model is executable; this allows configurator to quickly verify the correctness of component configurations. Moreover, model contains constraints on its parameters. These constraints are internal consistency or some external properties. Constraints are translated into asserts in generated source code. Therefore, we can check these constraints both at model simulation and at source code execution. This paper presents our approach to describe such models at Scala language. We successfully tested the approach in RTOS JetOS.

Keywords: components; modularity; RTOS; formal models; code generation

DOI: 10.15514/ISPRAS-2018-30(3)-10

For citation: Mallachiev K.A., Khoroshilov A.V. Building Modular Real-time software from Unified Component Model. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 135-148. DOI: 10.15514/ISPRAS-2018-30(3)-10

1. Introduction

Modern embedded operating systems support several CPU architectures and a lot of peripheral devices. OS contains many drivers to support numerous different hardware. Embedded OS are often designed for execution in a restricted environment, for example, with limited memory. Most of the drivers are not needed for correct OS execution on some specific board and spend valuable resources. Therefore, OS must support configuration to select drivers, which will execute on the target hardware.

Static OS configuration is used in cases when it is known in advance, on which hardware the OS image is going to be executed. Static means that configuration is performed on the host machine before OS loading to the target machine. The result of static OS configuration is the final image, which can be run on the target. Static configuration allows keeping final image small.

Typically, there are two roles taking part in the process of OS image building. The first role is a developer of whole OS or some driver. Developer implements his part in some programming language, writes documentation and provides support of source code and documentation. The second one is a system integrator who is responsible for correct OS configuration for specific task of specific board. Usually the system integrator does not change OS source code.

Besides simple selecting, which driver will be in the final OS image, many operating systems support finer tuning. For example, configuration allows selecting file system for each hard drive, or set IP address that will be used by network stack. These details are configured statically because for embedded OS and especially for safety-critical systems simplicity is more important than generality.

It is a natural desire to divide the operating system into isolated components, but not every part of the OS can be isolated. For example, OS core often is strongly coupled and might be divided into isolated components only if the core will be fully redesigned to support new architecture.

If we investigate configurations of the same OS on different boards, then we will see that there is the most variable part in the OS. We call this part *OS drivers*. OS drivers contain device drivers and some services such as network stack, file system, logging, etc. Our work aimed to support flexible configuration of OS drivers.

It is common that there are many vendors involved in building of OS drivers. When services or drivers are strongly coupled, their developers have to interact a lot. Therefore, splitting OS drivers into independent isolated components helps to simplify and accelerate development.

Component should interact with each other. Appearance of fixed interface between components would make component development easier. Moreover, fixed interface can make system flexible. Only connected components can interact, and only component with the same interfaces can be connected. System integrator is responsible for connection of the components.

Suppose that system integrator created a composition of the components, which describes how each component is configured and how components are connected. We call component-based system flexible if the system integrator can:

- modify configuration of the single component without modifying others,
- substitute component with another one of the same interface without modifying other components,
- add a new component between two other connected components without modifying any component configuration except the new one.
- add to composition a copy of existing component, and they should not disturb each other.

We are developing an embedded real-time operating system for civil aircraft computers called JetOS [1]. JetOS is ARINC-653 compliant and statically configured. Approaches presented in this paper are designed for JetOS. Since JetOS is a RTOS, we are focused on minimizing the overhead added by component-based system.

2. Related Works

Classical distributed component models like Enterprise JavaBeans, CORBA and CORBA Component Model [2, 3] define components and interfaces between them. These models allow substituting one component with another one if both have the same interfaces. Brokers dynamically change components configuration. This dynamic configuration is not suitable for embedded systems with static configuration.

Ideas to separate OS appeared long ago in microkernels. Microkernel architecture's [4, 5] primary goal is to separates OS into independent servers that could be isolated from each other. Servers interact through inter-process communication (IPC). IPC calls are typed and servers with the same interface can substitute one another. But there cannot be two servers with the same interface; therefore, this model is not suitable for our tasks too.

OS-Kit [6] and eCos [7] apply modularity benefits into OS development process. They provide a set of OS components, which are used as building blocks to configure an OS. For configuration, eCos uses the Component Definition Language (CDL), an extension of the existing Tool Command Language (Tcl) scripting language. Configuration is represented as feature tree with internal dependencies, group and feature constraints. Enabling of one component can lead to enable of whole components subtree. Components can have calculated value in configuration, which are calculated based on other configuration parameters. However, this is not enough for our task. Configurator cannot manage component connections and cannot add copies of the same component.

µC/OS-II kernel uses THINK component framework [8, 9]. THINK is an implementation of the FRACTAL component model that aims to take into account the specific constraints of embedded systems development. Component describes through its interface. Interaction between components is possible after establishment of *bindings* between their interfaces. Binding is a communication channel between two or more components. Binding can be created between components of a distributed system (RPC binding). This concept also does not allow having several copies of the same component in the composition.

VxWorks is a popular embedded operating system. VxWorks board support package (BSP) is divided into components. Components interfaces are declared in Component Description Language (CDL). Note that this CDL is different from the CDL used in eCos. BSP developer can construct BSP from existing component and can add their own components. However, this system is not flexible. For example, each component has fixed list of component names, with which it can interact.

We are not aware of any component-based model with the following set of features:

- static configuration;
- low overhead;
- flexible configuration (in all aspects described in the introduction);
- type checking of the connection, i.e. checking that connected components have the same interface.

3. Component-based Model

Our model is component-based. Component has state, which is changed during model execution, and configuration, which is immutable. Components can communicate with other components via ports. Port is a set of functions; there are two kinds of ports: input ports and output ports. Output port can be connected with input port. Set of port function signatures is called port type. Only input and output port of the same port type can be connected.

Each function of a component input port has an assigned handler inside the component. Call of output port function leads to the call of connected input port, which, in turn, calls the assigned handler. These calls are standard function call, or in other words synchronous call inside the same thread. Therefore, component loses control during output port call.

Thus, port call keeps the current thread. Threads cannot be created dynamically during model execution. Threads count is constant during execution.

If component needs an additional thread, then this should be explicitly specified in the model. These components are called *active*. Active components have special handlers, which are called periodically or once in the context of the new thread. We call these handlers the *activity handlers*.

In order to facilitate component reuse we introduce the concepts of a *component type* and a *component instance*. Each component type can have any number of instances. The components described above are close to component instances.

Component type contains types of component state and configuration, but not their values. Component type contains types and names of input and output ports, but not their connection. In addition, component type contains implementation of:

- component initialization function, which is called at start and is used to initialize state based on the configuration;
- handlers assigned with input ports, if component has any;
- activity handlers if component is active.

Instances have unique values of state and configuration. It is easy to see that concepts of component type and component instance are similar to terms “class” and “class object” respectively.

3.1 Component Developer View

Component developer designs component state structure, how it should be initialized base on configuration and how it is changed during execution. Developer chooses types of configuration parameters. Developer does not aware of specific configuration parameters values, but he can add constraints on the values. He designs component input and output ports and implements handlers for input ports. Component’s input and output ports restrict component developer’s knowledge about “outside world”. He does not know how many instances of his component will be created or how they will be connected.

Component developer’s definition of component types consists of two parts: component type specification and implementation. Specification contains:

- component type name
- component input and output port names and their types
- structure of component configuration
- component’s purpose description: how it should be configured and in which environment its input ports should be called.

The rest of the information is private for component and is considered as implementation part.

3.2 System integrator view

System integrator gets specification of all component types in the system. System integrator decides how many instances of each component should be created and how they should be connected for solution of the specific problem. For each instance, integrator sets its configuration values.

3.2 Simple example

Suppose that component developer created *Amplifier* component type. *Amplifier* has single input port “in” and single output port “out”. In addition, it has single configuration parameter “factor”. Components aim is to amplify input signal from “in” port by factor “factor” and put output to “out” port.

Suppose that the system integrator wants to pass signal from two sensors to a single actuator, but he should amplify signal from first sensor by factor of 2 and from second one by factor of 10. System integrator decides to use *Amplifier* component type. He does not worry about implementation, only interfaces matters to him. For simplicity, let us assume that all ports have the same type. Amplifier component type as seen by system integrator can be seen at Fig. 1

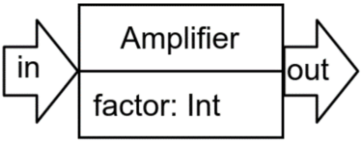


Fig. 1. Graphical representation of Amplifier component type specification

System integrator creates two instances of *Amplifier* component type: "amp1" with configuration value "factor" equal to 2 and "amp2" with configuration value "factor" equal to 10. Then connects them accordingly to sensors and to actuator. Scheme of the result can be seen at Fig. 2

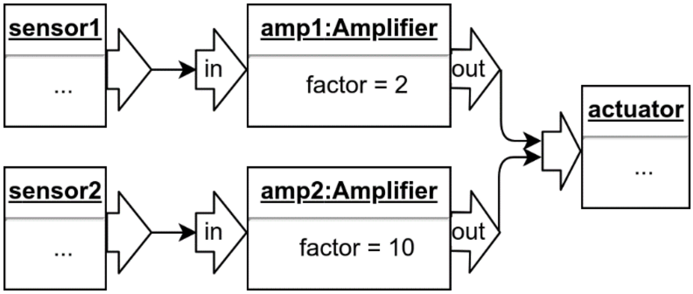


Fig. 2. Amplifier instances connection scheme

4. Prototype

In previous work [10], we implemented component-based approach in C language with some YAML code. We used common approach to apply object-oriented ideas in C language. Component state and configuration is presented as C structure, which explicitly passed to all component functions. Wrappers hid calls to output ports.

There was a lot of boilerplate code used to create component instances, describe their configuration, and their connections, in component type specification and its wrappers implementation.

To reduce amount of handwork we started to use YAML — simple declarative language. In the YAML developer specifies component type state, configuration, input and output ports, names of functions-handler for input ports. System integrator describes in the YAML component instances, their configuration and connections. We generated C code based on these YAML specifications.

This approach has some disadvantages.

- Component developer has to manually keep consistent two files (in YAML and C languages). Change in one file leads to change in another one.

- Component developer's workflow is not comfortable: after change in YAML code generation should be processed and only then C code should be updated accordingly.
- System integrator can connect instances incorrectly (this does not apply to type checking, which is performed during compilation) and cannot see the problem until final OS image is prepared and executed in target hardware.

5. Model-Based approach

We decided to go further along the path of abstraction and use abstract models of components and their composition. We use formal executable models. This has many benefits. Model contains more information than source code, thus source code can be generated based on the model. In addition, executable model allows simulating instances behaviour and their interaction. This is very useful for system integrator to quickly verify the correctness of configurations. Moreover, formal model can be used to formally verify its internal consistency.

We use Scala language to model components. Scala is a functional object-oriented language that suits us well.

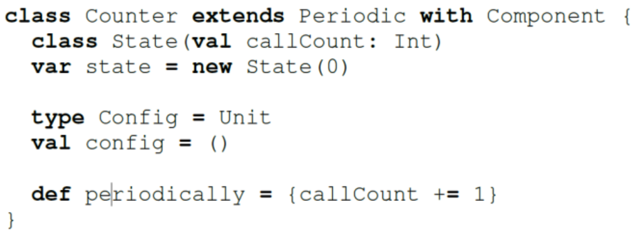
5.1 Model Description

5.1.1 Component Developer View

Component type is presented as Scala class inherited from interface (trait) «Component». Component configuration and state are the class fields with fixed names «config» and «state» respectively.

Active components have functions, which are called periodically or once. If component type inherits trait «RunOnce» then it should implement function «start», which will be called once after component initialization. If component type inherits class «Periodically», then it should implement function «periodically»; the frequency of the call is determined by the configuration.

For example, consider "Counter" component type, at Fig. 3, which has a state but no configuration. State contains value «callCount», which is initialized with zero. Function «periodically» increases «callCount» on every call.



```
class Counter extends Periodic with Component {
  class State(val callCount: Int)
  var state = new State(0)

  type Config = Unit
  val config = ()

  def periodically = {callCount += 1}
}
```

Fig. 3. «Counter» component type

Port types are declared as interfaces (traits). Input ports are defined inside component type class as objects, which inherited port type. Output port are class fields with type of port type. Output ports values are passed as component type constructor parameters. It is worth noting that output ports can be passed by name to constructor, this allows initializing component instances with cycle connections among them.

Example of input/output ports for “Amplifier” component type (defined in previous sections) can be seen at Fig. 4 Model can have constraints on state and configuration parameters values. These constraints are defined using Scala require function. Example of require statement for “Amplifier” component type can be seen at Fig. 5.

```
trait SignalProcessor {
  def processSignal(s:Int): Int
}

class Amplifier(out: =>SignalProcessor)...{
  ...
  object in extends SignalProcessor {
    def processSignal(s: Int): Int = {
      val processed = process(s)
      out.processSignal(processed)
    }
  }
}
```

Fig. 4. Port type SignalProcessor and ports of «Amplifier» component type. The component type has input port «in» and output port «out», both of them have type SignalProcessor. here is an implementation of function processSignal of «in» port. Port «out» passed-by-name. Scala syntax may be confusing, here function processSignal returns result of out port call

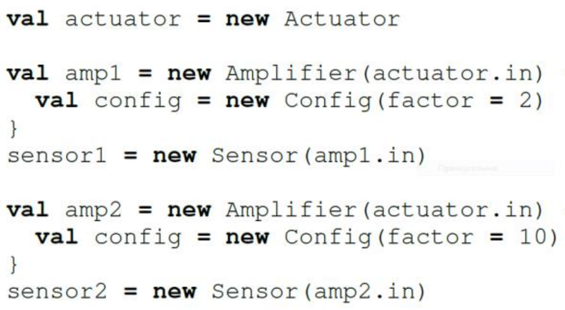
```
class Amplifier...{
  class Config(val factor: Int) {
    require (factor>0 && factor<50)
  } ...
}
```

Fig. 5. Configuration constraint for «Amplifier» component type; «factor» can take values only in the interval from 1 to 49

5.1.2 System Integrator View

System integrator creates instances of component type and connects them. For each instance, he defines its configuration parameters values.

As an example of component instances and their connections, consider model of the scheme depicted in the Fig. 2. This model can be seen at Fig 6.



```
val actuator = new Actuator

val amp1 = new Amplifier(actuator.in) {
  val config = new Config(factor = 2)
}
sensor1 = new Sensor(amp1.in)

val amp2 = new Amplifier(actuator.in) {
  val config = new Config(factor = 10)
}
sensor2 = new Sensor(amp2.in)
```

Fig. 6. «Amplifier» instances connection scheme

5.1.3 Preconfigured components

There is often component which have configuration parameters that have the same value in different configuration. To simplify configuration process for system integrator, we can define new component type, in which these parameters are fixed and cannot be configured. New component type class constructor calls constructor of the original one with values of these parameters. For example, it is possible to define "AmplifierBy2" which amplifies signal by fixed factor of 2.

It is more interesting to define new component, which is a composition of existing components. This is useful if some compositions are used often. Our approach assumes unified modeling of components and their composition. This allows using component-composition transparently for system integrator.

As an example, assume that there are component type «Amplifier» and «Filter», that are often connected. We create a new component type «AmplifyAndFilter» that is the composition of «Amplifier» and «Filter» Graphical representation of the «AmplifyAndFilter» component type can be seen at Fig. 7 and implementation at Fig. 8.

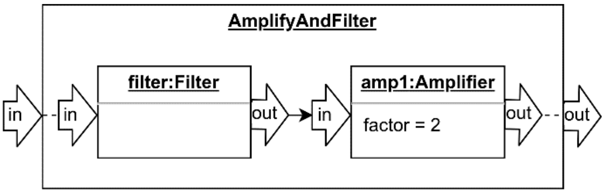


Fig. 7. Graphical representation of «AmplifyAndFilter» component type.

```
class AmplifyAndFilter(out: SignalProcessor)
    extends Component {
  val amp = new Amplifier(out) {
    val config = new Config(factor)
  }
  val filter = new Filter(amp.in)

  object in extends SignalProcessor {
    def processSignal(x:Int):Int = filter.in(x)
  }
}
```

Fig. 8. Implementation of «AmplifyAndFilter» component type

5.2 Model Usage

We use model to simulate instances behaviour and their interaction. We can verify that constraints are hold during simulation. In addition, we can write tests (unit and integration) to check that component model is correct.

We use model to generate C code, which gets into JetOS. We statically parse Scala code, extract needed information and translate it into C code.

Generated C code structurally looks much like code generated by prototype based on YAML files. We use same approach to model OOP in C language.

Some parts of the model can be translated into C without modifications, for example, simple operations and function calls. Some parts modified automatically during translation, but some can not be automatically translated without human help.

JetOS has strict coding style and, for instance, function can not have more than one `return` statement. We can generate code according this code style and, for example, we can automatically substitute several return statements in the model with a single one in the generated code.

As was mentioned, there are also statements, which cannot be easily translated into C. In addition, there are situations when generator tool cannot get enough information statically analysing Scala code. To solve these problems we add annotations to Scala code. Annotations does not change behaviour of model, they used only to provide additional information for the generator tool.

We use annotations to highlight input and output ports and their type interfaces. Annotations are «inport», «outport» and «interface» for input ports, output ports and port types respectively. As an example, Fig, 9 shows «Amplifier» component type with annotations.

```
@interface
trait SignalProcessor {
  def processSignal(s:Int): Int
}

class Amplifier(@outport out: SignalProcessor)...{
  ...
  @inport
  object in extends SignalProcessor {
    def processSignal(s: Int): Int = {
      val processed = process(s)
      out.processSignal(processed)
    }
  }
}
```

Fig. 9. Port type SignalProcessor and ports of «Amplifier» component type with annotations.

Scala language has rich syntax and not every statement can be easily translated to C. We allow annotating blocks of Scala code or Scala functions with C code. Fig. 10 contains partial example.

```
@C_code(code="int process(int* array) {...}")
def process(lst:List[Int]) = {...}
```

Fig. 10. C_code annotation example

This C_code annotation allows iteratively develop generator tool. At start, when tool supports only a few Scala statements, almost all code has C annotations. When support for new Scala statements adds to the tool, C annotations for these statements are no longer needed. Therefore, during tool development number of C_code annotations decreases.

6 Future Work

First, we still do not support many Scala statements and have a lot of C_code in our models. We are going to fix this in the new versions of generator tool.

For now, system developer should write Scala code by hand. This Scala code is very simple and matches a simple pattern. Thus, we can generate this Scala code from some GUI interface. Configuration constraints of the model can be extracted and added to this tool. This is one of optional future works.

Furthermore, formal model is a powerful tool and allows much more than C code generation. Formal model can be used for model checking and formal verifying internal consistency, preconditions or state invariants.

Tests and requirements can be generated based on the model and requirement generation is our next task. Requirement is the most important part of safety-critical system certification. Requirement writing is a hard handwork and automation (at least partial) will be very helpful.

7 Conclusion

The paper presents continuation of the work on modularity of RTOS. OS drivers are decomposed into isolated components. System integrator carries out component composition, and it can be done without contacting component developers and without writing C code.

We use a unified formal model to specify both components and their composition. Model, which is written in Scala language, is used to generated C code.

Also, model is executable, this allows system integrator to quickly verify correctness of composition. Model contains constraints on the model parameters. These constraints are tested during model simulation, also constraints can be translated into asserts in the generated C code.

Model-based approach still has disadvantage since the model is divided in two parts written in two languages, which have to be manually kept consistent. However, C code for some Scala statement is placed right before the statement, we hope that this will stimulate developers to update parts synchronously. Maturing of the generator tool decreases amount of C code in the model and reduces the importance of the problem.

The approach has been successfully tested on OS drivers of JetOS — ARINC-653 compliant RTOS. ARINC-653 has restrictions on the code executed in OS. For instance, resources (like buffers, semaphores, threads, etc.) can be requested only during initialization stage. Model restriction on threads creation apply well to ARINC-653 restrictions. Moreover, constructor code of the component type class is executed during initialization stage. Thus, component can request resources in the constructor.

References

[1]. K.M. Mallachiev, N.V. Pakulin, and A.V. Khoroshilov. Design and architecture of real-time operating system. Trudy ISP RAN / Proc. ISP RAS, vol. 28, no. 2, 2016, pp. 181–192. DOI: 10.15514/ISPRAS-2016-28(2)-12

[2]. J. Siegel and D. Frantz. CORBA 3 fundamentals and programming. John Wiley & Sons New York, NY, USA, 2000, vol. 2.

[3]. N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the corba component model. In Component-Based Software Engineering. Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 557–571.

[4]. A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach. In Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system, 2000, pp. 109–114.

[5]. I. Boule, M. Gien, and M. Guillemont. Chorus distributed operating systems. Computing Systems, vol. 1, no. 4, 1988, pp. 305-370.

[6]. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for kernel and language research. ACM SIGOPS Operating Systems Review, vol. 31, no. 5, 1997, pp. 38–51.

[7]. A. Massa, Embedded software development with eCos. Prentice Hall Professional Technical Reference, 2002.
[8]. J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In Proceedings of the USENIX Annual Technical Conference, General Track, 2002, pp. 73–86.
[9]. F. Loiret, J. Navas, J.-P. Babau, and O. Lobry. Component-based real-time operating system for embedded applications. In Proceedings of the International Symposium on Component-Based Software Engineering. Springer, 2009, pp. 209–226.
[10]. K. Mallachiev, N. Pakulin, A. Khoroshilov, and D. Buzdalov. Using modularization in embedded OS. Trudy ISP RAN / Proc. ISP RAS, vol. 29, issue. 4, 2017, pp. 283–294. DOI: 10.15514/ISPRAS-2017-29(4)-19

Построение модульного программного обеспечения на основе однородной компонентой модели

[1,2] *К.А. Маллачиев <mallachiev@ispras.ru>*
[1,2,3,4] *А.В. Хорошилов <khoroshilov@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25,
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1
[3] *Московский физико-технический институт,*
141700, Московская область, г. Долгопрудный, Институтский пер., 9
[4] *Высшая школа экономики,*
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Современные операционные системы реального времени являются сложным продуктом, разрабатываемым многими поставщиками: непосредственными разработчиками ОС, поставщиками пакета поддержки аппаратуры, разработчиками драйверов устройств и т.д. Такие ОС спроектированы так, чтобы иметь возможность запускаться на различном оборудовании, часто имеющем ограниченные ресурсы. Встраиваемые ОС содержат множество настроек и драйверов для поддержки разной аппаратуры. Большинство из этих драйверов являются излишними для запуска ОС на каком-то конкретном оборудовании. ОС статически конфигурируется для выбора набора драйверов и настроек для каждого типа аппаратуры. Модульность ОС упрощает как разработку ОС, так и ее конфигурирование. Разделение ОС на изолированные модули с фиксированными интерфейсами уменьшает необходимость взаимодействия между разработчиками в ходе совместной разработки. Мы используем формальные модели для описания компонентов и их взаимодействия. Использование формальных моделей приносит большую пользу. Описываемые модели содержат достаточно информации для генерации исходного кода компонента на языке Си. Предоставляемые модели являются исполняемыми, что позволяет человеку, отвечающему за конфигурацию, быстро проверить правильность заданной конфигурации. Кроме того, модель содержит ограничения на конфигурационные параметры. Примером таких ограничений являются ограничения на внутреннюю согласованность модели. При генерации исходного кода такие ограничения транслируются в специальные проверки на уровне исходного кода. Следовательно, ограничениями могут быть проверены как во время симуляции модели, так и во время исполнения исходного кода. В данной работе представлен подход к описанию таких моделей на языке программирования Scala. Мы успешно апробировали данный подход на основе ОС реального времени JetOS.

Ключевые слова: компоненты; модульность; ОСРВ; формальные модели; генерация кода

DOI: 10.15514/ISPRAS-2018-30(3)-10

Для цитирования: Маллачиев К.А., Хорошилов А.В Построение модульного программного обеспечения на основе однородной компонентой модели. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 135-148 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-10

Список литературы

[11]. K.M. Mallachiev, N.V. Pakulin, and A.V. Khoroshilov. Design and architecture of real-time operating system. Trudy ISP RAN / Proc. ISP RAS, vol. 28, no. 2, 2016, pp. 181–192. DOI: 10.15514/ISPRAS-2016-28(2)-12

[1]. J. Siegel and D. Frantz. CORBA 3 fundamentals and programming. John Wiley & Sons New York, NY, USA, 2000, vol. 2.

[2]. N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the corba component model. In Component-Based Software Engineering. Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 557–571.

[3]. A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The sawmill multiserver approach. In Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system, 2000, pp. 109–114.

[4]. I. Boule, M. Gien, and M. Guillemont. Chorus distributed operating systems. Computing Systems, vol. 1, no. 4, 1988, pp. 305-370.

[5]. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for kernel and language research. ACM SIGOPS Operating Systems Review, vol. 31, no. 5, 1997, pp. 38–51.

[6]. A. Massa, Embedded software development with eCos. Prentice Hall Professional Technical Reference, 2002.

[7]. J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In Proceedings of the USENIX Annual Technical Conference, General Track, 2002, pp. 73–86.

[8]. F. Loiret, J. Navas, J.-P. Babau, and O. Lobry. Component-based real-time operating system for embedded applications. In Proceedings of the International Symposium on Component-Based Software Engineering. Springer, 2009, pp. 209–226.

[9]. K. Mallachiev, N. Pakulin, A. Khoroshilov, and D. Buzdalov. Using modularization in embedded OS. Trudy ISP RAN / Proc. ISP RAS, vol. 29, issue. 4, 2017, pp. 283–294. DOI: 10.15514/ISPRAS-2017-29(4)-19