POK, an ARINC653-compliant operating system released under the BSD license

Julien Delange
European Space Agency
Keplerlaan 1, 2201AG Noordwijk, The Netherlands
julien.delange@esa.int

Laurent Lec
MakeMeReach
23 rue de Cléry, 75002 Paris, France
lec.laurent@gmail.com

Abstract

High-integrity systems must be designed to ensure reliability and robustness properties. They must operate continuously, even when deployed in hostile environment and exposed to hazards and threats.

To avoid any potential issue during execution, they are developed with specific attention. For that purpose, specific standards define methods and rules to be checked during the development process. Dedicated execution platforms must also be used to reduce potential errors. For example, in the avionics domain, the DO178-B standard defines the quality criteria (in terms of performance, code coverage, etc.) to be met according to the software assurance level. ARINC653 specifies services for the design of safe systems of avionics systems by using partitioning mechanisms.

However, despite those specific methods and tools, errors are still introduced in high-integrity systems implementation. In fact, their complexity due to the large number of collocated functions complicates their analysis, design or even configuration & deployment. In addition, an error may lead to a safety or security threats, which is especially critical for such systems.

In addition, existing tools and software are released under either commercial or proprietary terms. This does not ease identification and fix of potential security/safety issues while also reducing the potential users audience.

In this paper, we present POK, a kernel released under the BSD license that supports software isolation with time & space partitioning for high-integrity systems implementation. Its configuration is automatically generated from system specifications to avoid potential error related to traditional code production processes. System specifications, written using AADL models are also analyzed to detect any design error prior implementation efforts.

1 Introduction

Safety-critical systems are used in domains (such as military, medicine, etc.) where security and safety are a special interest. They have strong requirements in terms of time (enforcement of strong deadlines) and resources consumption (low memory footprint, no dead code, etc.). These systems must ensure a continuous operational state so that each potential failure must be detected before generating any defective behavior. However, they often operate in hostile environments so that they must be especially tested, validated to ensure absence of potential error.

Last years, processing power of such real-time embedded systems increased significantly so that many functions previously implemented using dedicated hardware are now provided by software, which facilitates updates and reduces development costs. In addition, this additional processing power gives the ability to collocate several functions on a same computation node, reducing the hardware to deploy and thus, the overall system complexity.

However, by collocating several functions on the same processor brings new problems. In particular, this integration must ensure that each function will be executed as it was deployed on a single processor. In other words, the execution run-time must provide an environment similar to the one provided by a single processor. In addition, impacts between collocated functions must be analyzed, especially for safety-critical systems where integrated functions may be classified at different security/safety levels.

Next section details the problem and our proposed approach. It also presents other work related to this topic.

2 Problems & Approach

The following paragraphs details identified problems related to high-integrity systems design and development with the increasing number of functions and their integration on the same processor. Then, it details our approach and each of its functionality.

2.1 Problems

High-integrity systems host an increasing number of functions. In addition, due to new hardware and performance improvements, the trend is to exploit additional computing capacity and collocate more software components on the same processor. However, this introduces new problems, especially when systems have to enforce safety or security requirements:

1. The platform must provide an environment for system functions so that they can be executed as if they run on a single processor.
2. Functions collocated on the same processor must be analyzed so that an error on a function cannot impact the others. For example, this would ensure that a function classified at a given assurance level cannot impact another at a higher one.
3. The execution platform configuration must be error-free to ensure functions isolation and requirements enforcement. Otherwise, any potential mistake would lead to misbehavior and make the whole system unstable.

First problem relates to function isolation: while collocating several functions on the same processor, designers must ensure their isolation. For that purpose, functions are separated within partitions. These are under supervision of a dedicated kernel that provides partitions execution separation using **time & space isolation**:

- **Time isolation** means that processing capacity is allocated to each partition so that it is executed as if it was deployed on a single processor.
- **Space isolation** means that each partition is contained in a dedicated memory segment so that one partition cannot access the segment of another. However, partitions can communicate through dedicated channels under kernel supervision so that only authorized channels are granted at run-time.

Partitions integration analysis is especially important since a partitioned kernel may collocate several functions classified at different security/safety levels. Communication channels between partitions must be verified to check that system architecture does not break safety/security requirements. For example, in the context of safety, we have to check that a partition at a low safety level cannot block a device used by a partition at a higher security level. On the other hand, on security level, we have to check that a partition cannot establish covert channel to read or write data from/to partition classified at higher security levels.

Thus, automatic kernel and partitions configuration would be a particular interest because these systems host partitions that provide critical functions and must be carefully designed. In consequence, we have to avoid any manual code production process, because it is error prone and have important economic and safety impacts [6]. This is also especially critical when creating the most important code - the one that configures the isolation policy.

2.2 Approach

To cope with identified problems, we propose the following approach focused on the following aspects:

1. **A dedicated run-time**, POK, that ensures time & space partitioning so that functions are executed as if they run on a single processor.
2. **An analysis framework** that analyzes system functions and detects potential errors.
3. **An automatic partitions and kernel configuration tool** from validated specifications.

This ensures enforcement of validated safety and security requirements at run-time.

Use of these three steps altogether makes a complete development process that would ease integration of heterogeneous functions, as illustrated in figure 1. First, the designer describes its functions with its properties (criticality/security levels, execution time, etc.) using an appropriate specification language (AADL). Then, we analyze system architecture to ensure that each of their requirements would be enforced while integrating (step 1 on 1). From this analysis, deployment & configuration code is automatically produced so that partitions and kernels are correctly configured according to their requirements (step 2 on 1). Finally, generated code is integrated with the POK execution platform that supports time & space isolation so that functions are integrated and separated as specified (step 3 on 1).

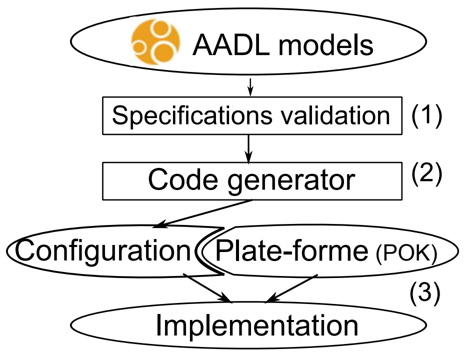

FIGURE 1: *Overall development approach*

As a result, our approach ensures integration of several functions on the same processor while preserving all their requirements in terms of timing, security or safety.

2.3 Related Work

Partitioning operating systems that support time & space partitioning already exist. However, most existing solutions are released as commercial and proprietary software, such as Vxworks [8] or LynxOS [9]. On the free-software side, the Xtratum [13] project aims at providing partitioning support by isolating functions in RTEMS [11] instances. However, it isolates functions using virtualisation mechanisms (using a dedicated hypervisor), whereas POK relies on a traditional kernel model.

In addition, several standardization attempts were initiated over the years, to define the concepts behind partitioned operating systems. Among them, the ARINC653 avionics standard [2] specifies the services and the API that would be provided by such a system. The MILS [14] approach, dedicated to security, also details required services to integrate several functions on the same processor while ensuring a security policy.

On our side, the POK operating system provides services required by both ARINC53 & MILS: time & space partitioning support, real-time scheduling, device drivers isolation, etc. It supports ARINC653 API and provides a POSIX adaptation layer to ease application use on POK. Finally, it is released under the BSD license so that users are willing to use it as free-software and can easily improve it, depending on their needs.

On system modeling and analysis side, no framework provides the capability to both capture and analyze partitioned architecture requirements. However, this is very important for incoming projects when system architecture must be analyzed/validated and development automated, especially because traditional development methods costs are still increasing [6] and lead to security/safety issues.

Similarly, no tool automates configuration and deployment of partitioned kernel from their specifications. Usually, developers make it manually by translating system requirements into configuration/deployment code. This is still error-prone and a fault can have a significant impact (missed timing constraint/deadline, communication channel not allowed, etc.). Automating configuration is still an emerging need but would likely to take importance while system functions are still increasing.

3 The POK execution platform

The following sections gives a general presentation of the POK kernel, detailing its services and implementation internals.

3.1 Overview

POK is an operating system that isolates partitions in terms of:

- **time** by allocating a fixed time-slice for partitions tasks execution with its own scheduling policy.
- **space** by associating a unique memory segment to each partition for its code and data.

However, partitions may need to communicate. For that purpose, POK provides the inter-partitions ports mechanism that defines interfaces to exchange data between partitions. These are defined in kernel configuration so that only specified channels are allowed at run-time. Consequently, system designer has to specify the communication policy prior (which partition can communicate with another) executing the system.

Moreover, partitions may require to communicate with the external environment and so, access devices. However, sharing a device between two partitions leads to potential security or safety issues: a partition at a low security level may read data previously written by a partition classified at a higher level. To prevent this kind of issue, POK requires that each device is associated to one partition and the binding between partitions and devices must be explicitly defined during the configuration.

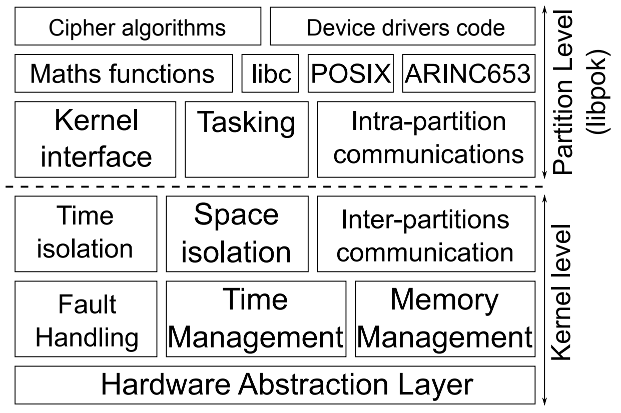

FIGURE 2: *Services of Kernel and Partition layers*

Finally, POK is available under BSD licence terms, supports several architectures (x86, PowerPC, SPARC/LEON) and has already been used by different research projects [15, 5, 18].

3.2 POK services

Figure 2 depicts the services of each layer (kernel/-partition). The kernel contains the following:

- The **Hardware Abstraction Layer** provides a uniform view of the hardware.
- **Memory Management** handles memory to create or delete memory segments.
- **Time Management** counts the time elapsed since system start-up and provides time-related functions to partitions (for supporting scheduling algorithms for example).
- **Fault Handling** catches errors/exceptions (for example: divide by zero, segmentation fault, etc.) and calls the appropriate handler (partition or task that generates the fault).
- **Time Isolation** allocates time to each partition according to kernel configuration.
- **Space Isolation** switches from one memory segment to another when another partition is selected to be executed.
- **Inter-partitions communication** exchanges data from one partition to another. Inter-partitions communication is supervised by the kernel so that only explicitly defined communications channels can be established and exchange data.

Partition-level services aims at supporting applications execution. It provides relevant services to create execution entities (tasks, mutexes, etc ...) and helps developers:

- **Kernel interface** accesses kernel services (writing to/reading from inter-partitions communication ports, creating tasks, etc.). It relies on software interrupts (syscalls) to request specific kernel services.
- **Tasking functions** handles task-related aspects (thread management, mutex/semaphore locking, etc.).
- **Intra-partition communication** provides functions for data exchange within a partition. It supports state-of-the-art communication patterns: events or data-flow oriented, semaphores, events, etc. Intra-partitions services are the same than the one provided in the ARINC653 [2] standard.
- **Libc, Maths functions, POSIX & ARINC653 layers** are the mapping of well known standards in POK. It aims at adapting other APIs to help developers for reusing existing code in POK partitions. For example, by using this compatibility layer, developers can reuse existing software that performs POSIX calls.
- **Device drivers code** is a set of functions to support devices within partitions. It relies on the kernel-interface service when privileged instruction (see section 3.5) are used.
- **Cipher algorithms** is a set of reusable functions to crypt/decrypt data within a partition.

3.3 Time isolation policy

The time partitioning policy requires a predictable method to ensure enforcement of time-slices allocation. For that purpose, POK schedules partitions using a round-robin protocol: each of them is executed periodically for a fixed amount of time. This introduces a scheduling hierarchy: partitions are first scheduled and then, tasks within the partition are scheduled according to the internal partition scheduling policy.

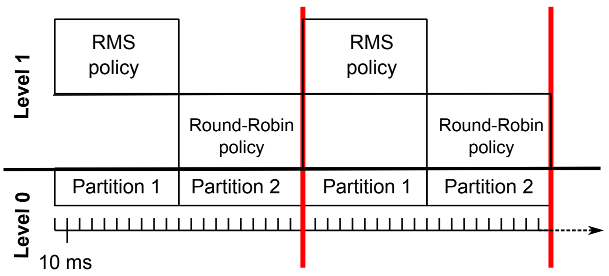

FIGURE 3: *Partitions scheduling example using POK*

In order to ensure predictable communications (in terms of time), inter-partitions communication data are flushed at the **major frame**, which is the end of the partition scheduling period (sum of all partitions periods). In other words, once all partitions are executed, inter-partitions communications data are flushed and is available to recipients partitions during their next execution period.

An example of such a scheduling policy is provided in figure 3 with two partitions: one schedules its task using the Rate Monotonic Scheduling (RMS) policy while the other uses Earliest Deadline First (EDF). Each partition is executed during 100ms and inter-partitions communication ports are flushed each 200ms.

Within partitions, POK supports several scheduling policies: the basic FIFO algorithm or other state-of-the-art methods such as EDF or LLF. However, support for advanced scheduling algorithms is still considered as experimental.

3.4 Space isolation and inter-partitions communication

Partitions are contained in a distinct memory segment, which means that each of them has a unique space to store its code and data. Segments cannot overlap each other and a partition cannot access to memory address located outside its associated segment.

Segments properties (address, size, allocation) are defined at configuration time and cannot be modified during run-time. To avoid any access outside their segments, partitions are executed in a *user* context. They can only use a restricted set of instructions[1], as for user processes on regular operating systems. To use privileged instructions[2], they must call kernel services and performs system calls (see POK services - kernel interface in 3.2).

To guarantee space isolation, POK relies on the *Memory Management Unit* (known as the *MMU*), a dedicated hardware component usually embedded in the processor. Depending on the architecture, different protection mechanisms can be used (segmentation, pagination, etc.). For the x86 platform, it relies on the segmentation. This protection pattern has many advantages: it is very simple to program (the developer has to define memory areas using the Global Descriptor Table - GDT [12]) and is predictable (other memory protection mechanisms requires to perform special operations which execution time are difficult to bound). When switching from a partition to another, the POK kernel ensures that:

1. Execution context from the previous partition is clean (the elected partition cannot read data produced by the previous).
2. The memory segment associated with the elected partition is loaded.
3. Caches are flushed so that the execution context is clean as if the partition was freshly elected on a clear environment.

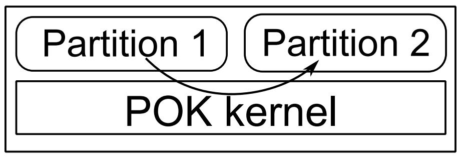

FIGURE 4: *Inter-partitions communication within POK*

Finally, POK provides the inter-partition communication services, which aim at establishing and monitoring data exchange from one partition to another, as shown in figure 4, where partition 1 send data to partition 2 under the supervision of the POK kernel. These channels must be defined at configuration time so that the kernel allows only explicitly defined communication at run-time.

[1] On Intel architectures, such code is usually executed in ring 3
[2] On Intel architectures, privileged instructions are available for code executed in ring 0

In the example of figure 4, any attempt to send data from partition 2 to partition 1 will fail. As explained in section 3.3, inter-partitions communication are flushed when the major time frame is reached. To do so, the POK kernel copies the data of each initialized source ports to their connected destination ports, copying memory areas from one partition to another. However, to ensure that only relevant data are copied, the size of each inter-partition port is also defined at configuration time so that no additional data can be read by the destination partition.

3.5 Device drivers implementation

Compared to other approaches, POK does not execute device drivers within the kernel. Instead, their code is isolated within a single partition to:

- Avoid any impact from a device driver error (*safety* reason). If the driver is executed in the kernel context, a crash would lead to unexpected impacts, such as crashing the whole kernel or other partitions.
- Ensure data isolation (*security* reason): if the device is shared by several partitions, one classified at a low security level may read or write data on the device from another one classified at higher level.

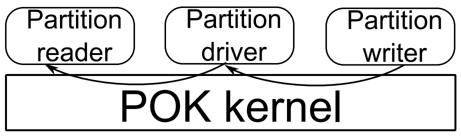

FIGURE 5: *Model for sharing a device using POK*

However, some partitions may need to share a device. Even if they cannot have their own driver instance, these partitions may communicate with the one that executes the driver. In that case, the connection between partitions must be explicitly defined at configuration time and then, be enforced by the security policy. This is illustrated in figure 5: the device that reads the device receives data from the partition driver while the writer has an outgoing connection to it. Connection to the driver must be explicitly defined so that the partition driver sends data only to relevant communication ports.

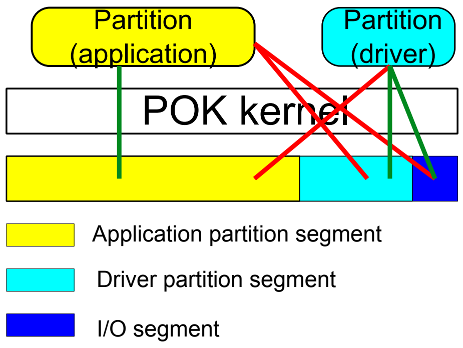

FIGURE 6: *Input/Output operations within POK*

Finally, when executing within a partition, a driver requires to have access to privileged instructions to control hardware. For that purpose, POK provides access to these privileged instructions only to the relevant partition (the one that accesses the driver). In that case, this additional access must be defined at configuration time so that the kernel grants/refuses access to privileged instructions according to its configuration policy. An example is shown in figure 6: in this system, two partitions are executed: one application partition and a driver partition that controls a device. The system defines three memory areas: one for each partition and an additional area mapped to the device memory. The driver partition may have access to this specific additional memory segment to control the hardware but the application partition would not be able to access it.

3.6 Configuration flexibility

One major advantage of POK compared to other partitioned operating systems is its fine-grained configuration policy: each service of POK, at each level (partition of kernel) can be configured and finely tuned. This would have several advantages: first, it reduces the set of functionality to the minimum and ensures a reduced memory footprint. In addition, reducing the number of functions and potentially useless code increases the code coverage of the whole system (for partitions and kernel).

However, this flexibility is particularly interesting when the system is automatically configured, according to its specification/description. Otherwise, the developer has to write the configuration manually, which would be time-consuming and error-prone. Hopefully, POK provides such configuration generation facility through its AADL code generator.

To auto-generate kernel and partitions configuration, designer must then first model its system with its requirements. This is explained in the next sections.

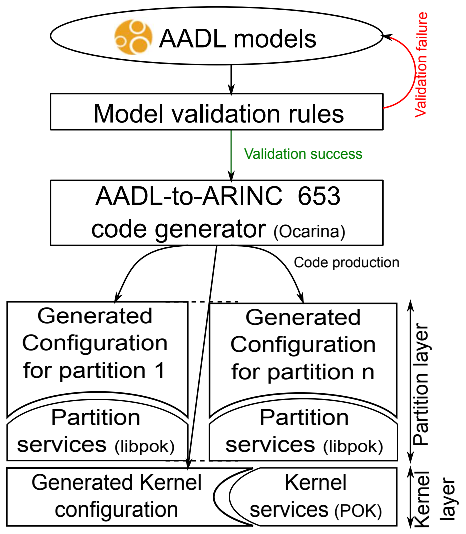
AADL models
Validation failure
Model validation rules
Validation success
AADL-to-ARINC 653 code generator (Ocarina)
Code production
Generated Configuration for partition 1
Partition services (libpok)
Generated Configuration for partition n
Partition services (libpok)
Partition layer
Generated Kernel configuration
Kernel services (POK)
Kernel layer

FIGURE 7: *Development and prototyping process using AADL models*

4 Partitioned System Modeling

To define a system architecture that uses POK, its specification is written with an appropriate modeling language to:

1. Specify system functions using a rigorous notation (better than a simple text-based document)
2. Analyze system properties and requirements to check for potential errors before implementation efforts. (for example, check system architecture against a security policy).
3. Generate configuration & deployment code using automated tools so that we avoid hand-written code and its related problems.

For our needs, we make a specific process (as illustrated in figure 7) for the design and implementation or partitioned systems supported by POK. First of all, engineers create AADL [7] models that specify system constraints and requirements. These specifications are processed by validation tools that check for potential error/fault that could generate a problem either at run-time or even for system implementation. Finally, code generator automatically creates configuration & deployment code for both kernel and partitions according to system requirements (AADL models) so that verified properties are correctly translated into code.

Next sections present the language chosen (AADL [7]), its tailoring for partitioned architecture modeling and its use for specifications validation.

4.1 AADL

The Architecture Analysis and Design Language (AADL) [7] is a modeling language that defines a set of components to model a system architecture. It provides hardware (`processor`, `virtual processor`, `memory`, etc.) and software components (`thread`, `data`, `subprogram`, etc.) as well as mechanisms to associate them (for example, model the association of a `thread` to a `processor` or a `memory`). The language itself can be extended using:

1. **Properties** to specify components requirements or constraints (for example, the period of a task). The core language provides a set of predefined properties but system designers can define their own.
2. **Annex language** to associate a piece of code of another language to a component. By doing so, it provides a way to interface AADL components with third-party language for the association of other system aspects.

AADL provides both graphical and textual notations. The graphical one is more convenient for system designers while the text-based one is easier to process by system analyzers/code generators. A complete description of the language, written by its authors, is available in [1].

4.2 Tailoring the AADL for partitioned architectures

The AADL provides a convenient way to describe both software and hardware architectures but one may need guidance to know how to use it to describe partitioned architectures and especially how to use AADL components to describe POK run-time entities (partitions, memory segments, etc.) and their requirements/constraints (partitions scheduling, memory segments size, etc.).

For that purpose, we design modeling patterns, which consist of a set of modeling rules to be used by system designers to represent partitioned architectures. The main idea is to restrict the set of components to be used in a model and which properties must be associated to the model in order to make a complete description of a partitioned architecture.

These modeling patterns has been first design for POK [15] and then, be standardized as an annex of the AADL standard [3]. Most important patterns define how to use AADL to model a partition (`process` bound to a `memory` and a `virtual processor` to model the partition, its allocated memory segment and its execution context), intra-partition communication (connections of AADL ports between `thread` located within the same `process`) or inter-partition communication (connection of AADL ports between `process` components). A full description of these modeling patterns is available in [15, 3, 5].

Next section presents our tools that check models to validate partitioned architectures requirements.

4.3 System validation

Once system architecture is specified using AADL models, analysis tools process models and check their compliance with several requirements/guidelines. In the context of POK, we design the following rules:

- **Modeling patterns enforcement rules (rules described in previous paragraph)**. This ensures that models are complete, with all required components/properties and they can be processed to configure the kernel and partitions with code generators.
- **Handling of potential safety issues rule**. This one checks that all potential error will be recovered and report to the user which error is not handled by a partition or a task. For example, if the designer didn't specify a recovery subprogram when a memory fault is raised while execution a partition, the analyzer will report an error.
- **Security analysis rules**. This aims at checking the architecture against a security policy. Depending on its own classification level and the one of the data they produce or receive, a partition may be compliant with a specific security policy. On the other hand, the security policy defines which operations are *legal* so that our tool can automatically checks for architecture compliance with them. Actually, our tool checks state-of-the-art security policies such as Bell-Lapadula [17] or Biba [16].

Other validation can be issued on the models to check either the correctness of the architecture (for example: resource dimensions with the analysis of memory segments size with respect to the associated partitions requirements - size of tasks size, etc.) or validation of a specific constraint (for example, partitions scheduling).

Once models are validated and the architecture considered as correct, our code generator, Ocarina, produces configuration & deployment code for both the kernel and its associated partitions. Next sections detail this process.

5 Automatic Configuration & Deployment of Partitioned Architectures

Once system requirements and constraints have been validated, the Ocarina AADL tool-suite processes models and generates configuration & deployment code for both kernel and partitions, as shown in figure 7. Next sections describe the process [5], highlighting its benefits regarding safety-critical systems needs (predictability, safety assurance, etc.) and constraints (code coverage, etc.).

5.1 Code generation Overview

Our code generator process [5] consists in analyzing AADL models, browsing its components hierarchy and, for each of them, generates appropriate code to create and configure system entities (as depicted in figure 7). The process creates the kernel configuration code (partitions time slots, memory segments assignment for each partition, etc.) and partitions configuration code (services to be used, required memory, intra-partition communication policy, etc.).

For example, when the code generator visits a `memory` component, it generates code to create a new memory segment. Then, it inspects the model to retrieve the associated partition and configure the kernel to associate the appropriate partition with this segment. Each AADL entity and property is used to produce system configuration code so that the resulting process creates a complete code, almost ready to be compiled. Next section explains which part of the code is automatically generated and which still requires some manual code to have a complete executable system.

5.2 Kernel and partitions configuration

At first, the code generator browses components hierarchy to create the kernel and partitions configuration code. For that purposes, it analyzes the following AADL components:

- `processor`: for the kernel configuration. The `processor` components specifies the time slots allocated for each partition so that they are translated into C code to configure the kernel time isolation policy.
- `virtual processor`: for partition run-time configuration. Based on the properties of this component, the code generator produces code that configures partitions services (need for memory allocation, POSIX or ARINC653 API support, etc.)
- `process` and its `features`: the `process` contains all partitions resources (`thread` and `data`) so that necessary amount of resources is allocated in the kernel and its partition. `process features` represent inter-partitions communication ports. When analyzing such entities, the code generator configures these ports and their connection within the kernel so that only communication channels specified in the model would be granted at run-time.
- `memory`: as it represents a memory segment, the code generator produces configuration code that instantiates a new memory segment in the main system memory and associates it to its bound `process` component (that corresponds to a partition). In consequence, at run-time, each partition will be associated with a memory segment that has the properties (address, size, etc.) specified in the model.

Once this configuration code has been generated, Ocarina also generates the behavior code of the system, the one executed by each task. Next section details this step.

5.3 Behavior code generation

The behavior part corresponds to the code that uses partitions resources to execute application functions. It consists in getting/putting data from/to the execution environment (for example, by using inter-partitions communication ports), calling application functions and managing tasks execution (put a task in the sleep mode when its period is completed, etc.).

As for the configuration code generation, the behavior code of application is generated from a predefined set of AADL components:

- A `thread` component specifies a task that potentially communicates using its `features`. For each `thread`, Ocarina generates code that gets the data from its `IN features`, performs the call sequence to its `subprogram` and sends produced output using its `OUT features`.
- A `data` component represents a data located in a partition, shared among its tasks and protected against potential race conditions using locking mechanisms (mutex, semaphore, etc.). For each task that uses the data, the code generator produces code that locks it, modifies it and finally releases it.
- A `subprogram` component references object code implemented either in C, Ada or even assembly languages. This is just a reference to low-level implementation so when visiting such component, the code generator creates a function call to the object code. Finally, it also configures the build system in order to integrate the object code in the partition binary.

Then, almost all the code of the system is produced by Ocarina. The user has to provide the application-level code, the one referenced by AADL `subprogram` components and automatically called by the behavior code generated for each partition. Next section details the benefits of this process with respect to high-integrity systems requirements.

5.4 Benefits of Code Generation

First of all, the use of such a process requires to specify system architecture using a modeling language, which makes the whole process more rigorous than just using a text-based specification document. Moreover, the process brings the following benefits:

1. **Early error detection**
2. **Syntax/semantic error avoidance**
3. **Specifications requirements enforcement**

By using validated models as a language source for system implementation, the development process detects specification errors at the earliest when such problems are difficult to track and usually detected during tests (at best) or production (at worst) phases. By identifying these errors prior to the implementation, we save a significant number of problems and save development costs.

Then, by automating code production with code generators such as Ocarina, we rely on established generation patterns that output the same block of code according a predefined AADL block. Use of such code patterns avoids all error related to hand-written code that usually introduce syntax/semantic errors that are difficult to track[3] and require code analysis tools and reviews to be found.

Finally, a particular interest is the enforcement of the specifications. Implementation compliance with the specifications is usually checked during manual code review. However, this is long, costly and also error-prone [6] since its relies on a manual inspection. By automating the code production from the specifications and by using code generation patterns, implementation code ensures specifications enforcement and so, would reduce development costs while improving system safety and robustness.

6 Case-Study

We illustrate the POK dedicated design process through a basic example with two partitions: one that sends an integer to another which do some processing and outputs its result.

6.1 Overview

To focus on the design process, we limit the behavior of the system to a basic example: the sender partition executes a task each second, increments an integer (starting from 0) and sends the result to the receiver partition using an inter-partition communication channel. Then, the receiver partition runs a task activated each 500ms that retrieves the value sent by the other partition and triggers a divide by zero, depending on the received value. This, the second partition would execute the dedicated handler that recover from the fault it generates.

System architecture is made of two partitions each one executed during a 500ms time slot (so that the major frame is 1 second) and stored in a segment of 120Kbytes. Each partition contains a periodic thread (either the *sender* or the *receiver*) that calls one subprogram (the one that sends or receives the integer). Partitions communicate through two queuing ports, connection with a inter-partition channel. Finally, the recovery policy requires to stop the kernel when an error is raised at kernel-level and restart other components (partition, task) when one of them triggers an error/exception.

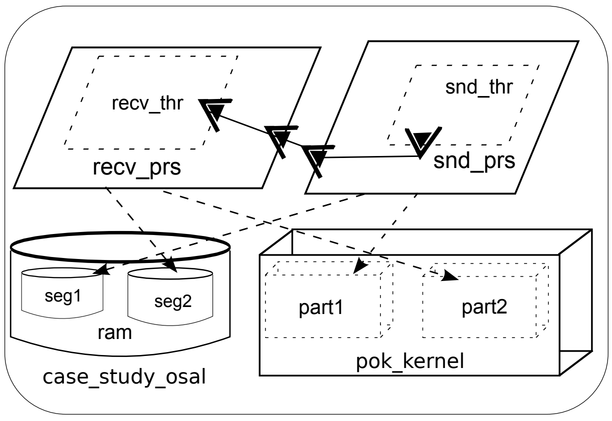

FIGURE 8: *AADL model of the case-study*

6.2 AADL modeling

The graphic version of the case-study is illustrated in figure 8. It maps the requirements previously described with specific execution components:

- **Partitions** use `process` (`recv_prs` and `snd_prs` for application aspects), `virtual processor` (`part1` and `part2` of `pok_kernel` for run-time concerns) and `memory` (segments specification) components. Arrows on the graphical model explicit their association.
- **Tasks** of each partition are defined with a `thread` component within each partition (either `recv_thr`) or `snd_thr`).
- **Queuing ports** are added on each partition and connected within their containing system (`case_study_osadl`)It explicitly defines communication channel that can be requested by partitions at run-time.

Definition of system recovery policy uses AADL properties, included only in the textual representation. The following listing provides an example of the definition of partition run-times with their recovery strategy (the `Recovery_Errors` and `Recovery_Actions` properties). It also includes a `processor` component that represents the partitioning kernel with the two partition environments (`part1` and `part2`) and its associated time isolation policy (`Slots`, `Slots_Allocation` and `Major_Frame` properties).

[3]for example, the C code `if (c = 1) statement;` is often an error even if legal and would likely be `if (c == 1) statement;`

```
virtual processor implementation partition.common
properties
   Scheduler => RR;
   Additional_Features=> (console, libc_stdio);
   Recovery_Errors     => (Illegal_Request,
                           Partition_Init);
   Recovery_Actions    => (Partition_Restart,
                           Partition_Stop);
end partition.common;

processor implementation pok_kernel.i
subcomponents
   part1 : virtual processor partition.common;
   part2 : virtual processor partition.common;
properties
   Major_Frame              => 1000ms;
   Slots                    => (500ms, 500ms);
   Slots_Allocation    =>
      (reference (part1), reference (part2));
   Recovery_Errors     => (Kernel_Init,
                                Kernel_Scheduling);
   Recovery_Actions    => (Kernel_Stop,
                                Kernel_Stop);
end pok_kernel.i;
```

The following listing also illustrates the definition of the complete system with its processor, partitions, memory and the connection of the inter-partitions queuing ports.

```
system implementation osal.i
subcomponents
   cpu      : processor pok_kernel.i;
   sender   : process   snd_prs.i;
   receiver : process   recv_prs.i;
   mem      : memory    ram.i;
connections
   port sender.pdataout -> receiver.pdatain;
properties
   actual_processor_binding =>
      (reference (cpu.part1)) applies to sender;
   actual_processor_binding =>
      (reference (cpu.part2)) applies to receiver;
   actual_memory_binding =>
      (reference (mem.seg1)) applies to sender;
   actual_memory_binding =>
      (reference (mem.seg2)) applies to receiver;
end osal.i;
```

6.3 AADL model validation

We validate system architecture using the Requirements Enforcement Analysis Language (REAL [19]) tool. It analyzes AADL components hierarchy and validates it against theorems. By using this tool, we validate model structure and compliance with modeling patterns. We illustrate that using two theorems.

The first one (see listing below) checks compliance of system architecture regarding space isolation and memory segments definition. It processes `memory` components contained in the main system (that corresponds to the main memory - like RAM) and ensures that each one contains `memory` subcomponents to specify memory segment with their size (by checking the definition of the `Word_Count` property).

```
theorem Contains_Memories
  foreach s in system_set do
    mainmem := {y in Memory_Set |
                 is_subcomponent_of (y, s)};
    partitionsmem :=
                {x in Memory_Set |
                 is_subcomponent_of (x, mainmem)};

    check ((Cardinal (mainmem) > 0) and
           (Property_Exists
             (partitionsmem, "Word_Count"))
          );
end Contains_Memories;
```

Second validation theorem (see listing below) checks the major frame compliance (see section 3.3 for its definition) with partitions time slots. To do so, the theorem processes each `processor` component of the system and checks that the value of the `Major_Frame` property is equal to the sum of the `Slots` value.

```
theorem scheduling_major_frame
  foreach cpu in processor_set do
    check (property_exists(cpu, "Major_Frame") and
           ((float (property (cpu, "Major_Frame")) =
           sum (property (cpu, "Slots"))))));
end scheduling_major_frame;
```

Other validation theorems can be designed and added to the process to automatically check for enforcement of other requirements or also verify specific modeling patterns. Interested readers may refer to the POK distribution available on the official POK website [10]: it contains a complete REAL [19] theorem library to check all modeling patterns related to partitioned architectures.

6.4 Code generation & metrics

Configuration and deployment code is automatically generated from AADL specifications. As models have been previously validated, produced output is expected to enforce system requirements.

Among all generated files, one especially important is `deployment.h`, which defines constant and macro that configure kernel services and set resources dimensions (amount of ports, partitions, etc.). The following listing provides an overview of the file generated from the model of this case-study.

```
/*
 * other configuration directives
 * ...
 */
#define POK_NEEDS_CONSOLE 1
#define POK_CONFIG_NB_PARTITIONS 2
#define POK_CONFIG_SCHEDULING_SLOTS {500,500}
#define POK_CONFIG_SCHEDULING_SLOTS_ALLOCATION {0,1}
#define POK_CONFIG_SCHEDULING_NBSLOTS 2
#define POK_CONFIG_SCHEDULING_MAJOR_FRAME 1000
#define POK_CONFIG_NB_PORTS 2
```

We can check and verify that configuration directives enforces model requirements: two partitions are defined, scheduling slots of each partitions (500ms) is correctly mapped, as well as the major frame (1s).

The code generation process not only configures the kernel but produces partitions configuration and behavior code. Developers only have to write application code, which corresponds to the functional part of the system. In this case-study, it consists of two functions: one that outputs an integer and stores it as a function argument (the one used on the sender side) and another that takes one integer as argument and process it (receiver side). The code provided by the developer is shown in the following listing, demonstrating that code production automation reduces manual code production activities.

In the following application code, the receiver part raises a division by zero exception when result of $(t + 1)\%3 == 0$ (line 16 of the application code). According to the recovery policy, when such a condition is met, the partition restarts. To show graphically that the partition is correctly restarted, we also output the number of times the function is executed using variable `step`. Its initial value is stored in the data from the partition binary so that when reloading the partition, the initial value is set again in the variable.

```
 1 void user_send (int* t)
   {
 3    static int n = 0;

 5    printf ("Sent value %d\n", n);
      n = n + 1;
 7    *t = n;
   }
 9
   static int step = 0;
11
   void user_receive (int t)
13 {
      int d;
15
      d = ( t + 1) % 3;
17    printf ("Step %d\n", step++);
      printf ("Received value %d\n", t);
19    printf ("Computed value %d\n", t / d);
   }
```

Generated application is compiled for Intel (x86) architecture and produces the following output during execution:

```
...
Step 3
Received value 5
[KERNEL] Raise divide by zero error
Step 0
Received value 0
Computed value 0
Sent value 8
...
```

One may notice that when the faulty condition of the application code ($(t + 1)\%3 == 0$, line 16 of the user application code) is reached (in that case, when receiving value 5), the receiver partition is restarted. Initial value of variable `step` (printed in the line `Step N`) is set back to 0, showing that the partition binary has been re-loaded.

To assess the memory consumption of generated systems, we also report generated kernel and partitions sizes (see table below). Partitions size is similar: they contain the same functionality and differ only by their application code. Both of them have a small size: 11kB for a complete system that embeds run-time functions for the support of user application. This demonstrates the lightweight aspect of the approach. Kernel size is also very small, especially for such a system that provides critical functions regarding safety and security issues.

Component	**Size**
Kernel	26 kB
Partition 1	11 kB
Partition 2	11 kB

7 Conclusions & Perspectives

This article presents POK, a BSD-licensed operating system that supports partitioning with time & space isolation. It also provides layers to ease deployment of existing code that uses established standards such as POSIX or ARINC653.

Beyond the operating system itself, POK relies on a complete tool-chain to automate its configuration & deployment and ease partitioned systems development. It aims at specifying system architecture and properties using a modeling language, AADL and verifying its requirements using dedicated analysis tools that process these specifications. Then, from this validated specifications, our tool-chain automatically generates code that configures/deploys kernel/partitions and execute application code provided by the user. This ensures specifications requirements enforcement and avoid all errors related to usual development process.

7.1 Perspectives

The domain of partitioned architecture is still emerging and there is many potential open perspectives. On the kernel side, there is a need for more hardware support (devices, architectures, etc.) and a wider support of existing standards, as for the ARINC653 layer (for example, to support the second part of the standard).

On the modeling and analysis part, there is a strong need to connect AADL models with other system representation or specifications. In particular, the production of AADL models could be automated from text-based specifications and model components/entities could be associated to external specifications such as DOORS. This would ease requirements traceability, which is a special interest for the design of high-integrity systems, when designers have to ensure that high-levels requirements are correctly mapped in the implementation.

References

[1] *Peter H. Feiler, David Gluch and John Hudak.* The Architecture Analysis and Design Language (AADL): An Introduction. Technical report, 02 2006.

[2] *Airlines Electronic Engineering.* Avionics Application Software Standard Interface. Technical report, Aeronautical Radio, INC, 1997.

[3] SAE Architecture Analysis and Design Language (AADL) Annex Volume 2

[4] *Fabrice Bellard.* Qemu, a fast and portable dynamic translator. In ATEC 05: Proceedings of the annual conference on USENIX Annual Technical Conference, pages 4141.

[5] *Julien Delange, Laurent Pautet and Fabrice Kordon.* Code Generation Strategies for Partitioned Systems. In 29th IEEE Real-Time Systems Symposium (RTSS08), pages 5356, Barcelona, Spain, December 2008. IEEE Computer Society.

[6] National Institute of Standards and Technology (NIST). The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, 2002.

[7] SAE. Architecture Analysis & Design Language v2.0 (AS5506), September 2008.

[8] Windriver VxWorks - `http://www.windriver.com`

[9] LynuxWorks LynxOS - `http://www.lynuxworks.com/rtos/`

[10] POK - `http://pok.safety-critical.net`

[11] RTEMS - `http://www.rtems.com`

[12] Global Descriptor Table - Wikipedia

[13] Xtratum - `http://www.xtratum.org`

[14] *John Rushby* - MILS Policy Architectures

[15] *Julien Delange, Laurent Pautet and Peter Feiler.* Validating safety and security requirements for partitioned architectures. In 14th International Conference on Reliable Software Technologies - Ada Europe, June 2009

[16] *Biba, K.J.* - Integrity considerations for secure computer systems. Technical report, MITRE

[17] *Bell, D.E., LaPadula, L.J.* - Secure computer system: Unified exposition and multics interpretation. Technical report, The MITRE Corporation (1976)

[18] *Julien Delange* - Intégration de la securité et de la sureté de fonctionnement dans la construction d'intergiciels critiques - PhDThesis

[19] *Olivier Gilles and Jérôme Hugues - Validating requirements at model-level* in *Ingnierie Dirige par les modles (IDM08)*