Towards Robust OSes for Appliances: A New Approach Based on Domain-Specific Languages

Gilles Muller, Charles Consel, Renaud Marlet,
Luciano Porto Barreto, Fabrice Mérillon, Laurent Réveillère
COMPOSE group, `http://www.irisa.fr/compose`
IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
{`muller,consel,marlet,lportoba,merillon,lreveill`}`@irisa.fr`
`tel:+33.2.99.84.72.87, fax:+33.2.99.84.71.71`

Appliances represent a quickly growing domain that raises new challenges in OS design and development. First, new products appear at a rapid pace to satisfy emerging needs. Second, the nature of these markets makes these needs unpredictable. Lastly, given the competitiveness of such markets, there exists tremendous pressure to deliver new products. In fact, innovation is a requirement in emerging markets to gain commercial success.

The embedded nature of appliances makes upgrading and fixing bugs difficult (and sometimes impossible) to achieve. Consequently, there must be a high level of confidence in the software. Additionally, the pace of innovation requires rapid OS development so as to match ever changing needs of new appliances.

To offer confidence, software must be highly *robust*. That is, for a given type of appliance, critical behavioral properties must be determined and guaranteed (*e.g.,* power management must ensure that data are not lost). Robustness can be provided by mechanisms and/or tools. The ideal approach takes the form of *certification* tools aimed at statically verifying critical properties. Such tools avoid the need for a laborious and error-prone testing process.

To be first in a market requires not only that the testing process be shortened, but the development time as well. To achieve this goal, three strategies are needed: *re-use* of code to rapidly produce a new product by assembling existing building blocks, factorization of *expertise* to capitalize on domain-specific experience, and *open-endedness* of software systems to match evolving functionalities and hardware features.

In this paper, existing OS approaches are assessed with respect to the requirements raised by appliances. The limitations of these approaches are analyzed and used as a basis to propose a new approach to designing and structuring OSes for appliances. This approach is based on *Domain-Specific Languages* (DSLs), and offers rapid development of robust OSes. We illustrate and assess our approach by concrete examples.

1 Existing OS Approaches

An OS (or an OS sub-system) conventionally consists of two levels: mechanisms and policies. Mechanisms can take the form of libraries [7] or abstract machines [8]. Policies correspond to algorithms based on mechanisms. Ideally, a policy should be implemented as code gluing mechanisms together. In practice, the separation between these two levels is not systematically achieved. This lack of separation makes it difficult to understand and reason about the behavior of the OS, which, in turn, compromises robustness. Furthermore, such a blurred separation makes it hard to identify building blocks which causes poor code re-use. Also, because building blocks are not clearly exposed, code expertise cannot be fully exploited. Factorizing expertise is made even harder by the use of a general-purpose, usually low-level, programming language which does not make domain-specific knowledge explicit. This situation puts limitations on robustness, code re-use, and OS expertise. Let us examine each of these limitations in turn.

1.1 Limited Robustness

OS designers have traditionally limited their view of robustness to isolation of system components. In this view, a component may consist of both a policy and its associated mechanisms, or it may represent a single policy, or a set of related mechanisms. Component isolation is implemented by boundary protection relying on hardware capabilities (*e.g.,* MMU) or code instrumentation [23]. Another approach is to use high-level, type-safe language that guarantees correct memory access; examples include Modula-3 used in Spin [3] and ML used in the Fox Net [9]. However, the compilation technology for such languages is at a level where the performance of generated code does not yet compare to the code produced by a C compiler. Furthermore, developing an OS requires low-level expressiveness, not covered by a high-level language (*e.g.,* manipulation of raw data). This situation often necessitates additional work in programming languages to develop various extensions [9].

Recently, a static approach aimed at verifying predefined safety rules of binary code has been proposed by Necula and Lee [15]. In addition to being static, their approach goes beyond memory isolation. For example, it enables one to verify the quantity of resources used by a program. Its main limitation is that it assumes that programs are written in a general-purpose language; thus, some properties of interest may be undecidable in general. To circumvent this problem, programmer assistance is required at various stages.

Microsoft has proposed a tool to improve driver quality which combines static and dynamic strategies to expose driver errors. These strategies include allocation fault injection and parameter verification [1].

The common limitation of the above approaches is that they do not assume a software architecture where separation between mechanisms and policies is explicit; as a result, no specific reasoning strategy can be applied to either layer.

1.2 Limited Code Re-Use

The software architecture of a system also plays a key role in code re-use. Structuring a software system in terms of components is now a well-recognized strategy to achieve code reuse. Notably, in the OS field, this structuring technique has lead to micro-kernel architectures [2, 13, 18]. In a micro-kernel architecture, each component (or server) corresponds to a domain boundary. However, communication across a domain introduces overhead. Because the software components are directly mapped into protected entities, their granularity impacts system performance. The OS architect thus faces the dilemma between defining fine-grain components to expose code re-use opportunities and introducing coarse-grain components to optimize performance. As shown by the Workplace project at IBM, a compromise cannot necessarily be reached [6].

Recently, several research projects have aimed at developing extensible OSes [3, 5]. Such OSes do not rely on hardware protection boundaries; instead, they use either strongly-typed languages [11] or software-fault isolation [23]. Extensible kernels consist of fine-grain components which enable low-level functionalities to be exposed. This system architecture drastically improve code re-use. Although extensible OSes provide an effective solution to code re-use, they do not address expertise re-use. In fact, low-level kernel mechanisms require detailed expertise, which does not necessarily correspond to the skills of the industry programmer.

1.3 Limited Expertise Re-Use

Even when mechanisms have been defined at an appropriate level and granularity, expertise is still required to implement policies. Indeed, kernel mechanisms are often highly parameterized to cover a large set of needs, and are also poorly documented. Any incorrect invocation likely leads to unexpected behavior. For example, no requirements on resource allocations are made explicit such that resource leaks can be prevented. Furthermore, the combination of several mechanisms must follow precise rules that are rarely explicitly documented.

This situation requires the policy programmer to master the kernel mechanisms. This expertise can, unfortunately, only be gained by careful examination of the source code and laborious debugging. In fact, few programmers reach a sufficient level of expertise so as to rapidly develop correct kernel code. In the context of appliances, the variety of products increases the demand for such experts. Finally, the lack of tools to assist developers in verifying mechanism usage makes the demand for experts even more critical.

1.4 Limited Extensibility

Appliances typically form a family of products that evolve over time. Even when expertise has been gained during a product's development, turn-over is so high in the computer industry that this expertise may not be retained. As a result, new product generations may require the same expertise to be re-acquired.

Besides expertise, extensibility critically depends on the OS architecture. Specifically, if the building blocks have not been clearly staged, policies and mechanisms may be intertwined. As a result, it becomes difficult to extend either mechanisms or policies. Finally, existing approaches to extending OSes enable new components to be added via modules or servers [3, 18]. However, other than memory protection, they do not offer any guarantees regarding the behavior of the added components. This limitation can have disastrous consequences considering the widespread nature of appliances.

2 A New Approach Based on Domain-Specific Languages

The common thread of our approach to designing OSes is Domain-Specific Languages (DSLs) [12]. In this approach, a DSL is developed for each family of sub-systems.

An overview of our approach. A DSL consists of two distinct parts: an abstract machine and a compiler from the DSL to the abstract machine. This structure enforces a two-level design: policies are written in the DSL, while the abstract machine is directly mapped into mechanisms. The two-level approach forces the designer to stage the design issues: the first step is aimed at characterizing the policies needed for the target sub-system family; the second step consists of determining the mechanisms that are common to these policies. The library of mechanisms is then used to define the abstract machine.

This policy characterization defines the program patterns needed to express the policies of interest. In addition, properties that are critical to the family of sub-systems are identified (*e.g.,* termination, resource allocation, . . .). Both program patterns and properties are used to design a language dedicated to writing the target policies. Program patterns suggest specific syntactic abstractions, while properties lead to specific language restrictions that make the properties decidable. The latter feature contrasts with General Purpose Languages (GPLs) where expressiveness is traded for verification. Importantly, the development of a new DSL rarely means the introduction of a brand new syntax. Rather, this process usually consists of restricting an existing language and adding domain-specific constructs and values.

Besides improving robustness, a DSL can also be used to expose information that can trigger domain-specific optimizations. For example, a parameter passed across layers need not be copied if the language guarantees that a policy only reads it; such optimizations are performed by the Flick IDL compiler [4]. Finally, our experience has shown that the restricted nature of DSLs drastically improves the development time of compilers, reduces the number of concepts to treat, and enables the production of high-quality code.

Our DSLs. Our approach has been carried out in practice on two families of sub-systems: *device drivers* and *active networks*.

The device driver study has lead to the design and implementation of two languages GAL [22, 17] and Devil [14]. GAL specifically targets graphics cards; from a high-level specification a compiler generates a complete device driver. Devil covers all types of devices and can be seen as an IDL for hardware programming; from a device specification the compiler generates low-level code to operate the device. Both DSLs offer high-level abstractions to overcome the intricacies of interaction with hardware, such as error-prone bit manipulations. They enable critical properties on device specifications to be verified; for example, writing a Devil specification is up to 5.9 times less prone to errors than writing equivalent C code [17]. Also, they drastically improve productivity; for example, a GAL specification is 10 times smaller than the corresponding X11 C driver. Finally, both GAL and Devil have demonstrated that DSLs can compete with equivalent C code [14, 22].

Our study of active networks [24] has lead to the development of a DSL called PLAN-P [20, 21]. This DSL allows application-specific protocols to be written and dynamically deployed on both routers and terminal equipment such as appliances. A network infrastructure is a shared resource, therefore application-specific protocols must be well-behaved. Consequently, PLAN-P has been designed such that properties guaranteeing the network safety be preserved (*e.g.*, termination, linear packet duplication, ...). In practice, protocol implementations in PLAN-P have been shown to be 3 times smaller than and as efficient as the equivalent C code [20, 21].

Let us now examine in detail in what ways a DSL-based approach improves robustness, code re-use, and OS expertise.

2.1 Improved Robustness

The explicit separation between mechanisms and policies enforced by the DSL approach enables specific reasoning about each level. Indeed, unlike GPLs, the expressiveness of a DSL is defined such that only well-behaved policies can be written. Compilation of policies address both static and dynamic verification of mechanism usage (*i.e.,* abstract machine usage). Statically, the compiler ensures that mechanisms are invoked with proper parameters and follow precise usage rules. If the DSL designer considers that a particular static verification places too large a burden on the language's expressiveness, dynamic checks can be emitted by the compiler. For example, in Devil enumerated types permit to precisely define the set of valid commands over a register. Run-time assertions (in a debug mode) enforce correct usage of the generated interfaces by the C programmer.

As a result, since the verification of DSL programs is intrinsic to the language design, the robustness of policies can be certified. In fact, these guarantees make DSLs a key technology for an innovation-intensive domain such as appliances.

2.2 Improved Code Re-Use

Code re-use is also intrinsic to our approach since a DSL, and its associated abstract machine, target a specific family of sub-systems. The common building blocks are clearly identified, and their use in implementing a policy is guaranteed by the DSL compiler. This process ensures full re-use of code, in contrast with libraries where re-use depends on programmer knowledge.

In traditional OS, improving re-use consists of opening mechanisms at a low level to enable a large variety of policies to be defined. As the mechanisms expose lower level functionalities, the programmer needs to introduce more glue code to set up the appropriate invocation context. In the context of DSLs, the level at which mechanisms are exposed is not an issue anymore, because it is the compiler that generates the code to set up the invocation context.

2.3 Improved Expertise Re-Use

As we have discussed previously, expertise on low-level mechanisms can be gained as appliances are developed. However, this expertise is not made explicit and can be lost as programmers change assignments. In our approach, expertise, in the form of implementation knowledge and safety rules, is captured by the DSL compiler. Accordingly, programmers need not to be domain experts. In practice, the DSL compiler corresponds to an expertise repository. As an example, our current PLAN-P run-time system relies on Solaris streams library which is known to be difficult and error-prone to use. The PLAN-P compiler fully hides this complexity from the programmer. In fact, non kernel-expert students have been able to develop PLAN-P protocols within a day.

The DSL approach represents a framework for policy development: it enables the programmer to focus on a policy algorithm and abstracts away implementation details. As a result, more attention is devoted to the functionalities of policies. In addition, experimenting with policies and tuning them become easier.

2.4 Improved Extensibility

The two-level design enforced by the DSL approach allows extensions to be made at two conceptual levels: policies and mechanisms. Extensions at the policy level are characterized by the DSL syntax and property verification. Extensions at the mechanism level can be achieved by changing the abstract machine implementation.

Active networks represent an outstanding example of an extensible system: protocols are dynamically deployed on a heterogeneous infrastructure (*e.g.*, routers, workstations, and appliances). Our work on PLAN-P not only enables protocols to be introduced as policies, but it also allows different abstract machine implementations to be developed on various hardware platforms.

3 Conclusion

We have proposed a methodology to design and implement a new generation of OSes for quickly evolving domains such as appliances. Developing OSes for such domains puts tremendous stress on robustness, code re-use, expertise re-use and extensibility. We have showed how the DSL approach represents a new solution to these issues. Our DSL approach has been validated on various families of sub-systems. These DSLs have successfully addressed these issues without a loss of efficiency.

Our goal is to systematize the use of DSLs to design and develop an embedded OS from scratch. In this new OS, each service will be captured by a DSL. We are currently analyzing families of sub-systems to determine the critical properties of each domain. In a future stage, we plan to conduct comparative studies with existing embedded OSes to assess the benefits and drawbacks of our approach.

References

[1] Using driver verifier to expose driver errors. http://www.microsoft.com/hwdev/driver/driververify.htm.

[2] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *1986 Summer Usenix Conference*, pages 93–112, 1986.

[3] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP'95 [19], pages 267–283.

[4] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, USA, June 15–18, 1997.

[5] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP'95 [19], pages 251–266.

[6] B.D. Fleisch. The failure of personalities to generalize. In HOTOS'97 [10], pages 8–13.

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, pages 38–51, St-Malo, France, October 1997.

[8] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In OSDI'96 [16], pages 137–151.

[9] R. Harper, P. Lee, and F. Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1998. (Also published as Fox Memorandum CMU-CS-FOX-98-02).

[10] *6th Workshop on Hot Topics in Operating Systems*, Cape Cod, Ma, May 1997. IEEE Computer Society.

[11] W.C. Hsieh, Fiuczynski M.E., Garrett C., Savage S., Becker D., and Bershad B.N. Language support for extensible operating systems. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, pages 127–133, Tucson, AZ, USA, February 1996.

[12] D.A. Ladd and J.C. Ramming. Programming the Web: An application-oriented language for hypermedia service programming. In *Fourth International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[13] J. Liedtke. On μ-kernel construction. In SOSP'95 [19], pages 237–250.

[14] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000. To appear.

[15] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In OSDI'96 [16], pages 229–243.

[16] *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[17] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. A DSL approach to improve productivity and safety in device drivers development. In *Proceedings of the* 15^{th} *IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000. IEEE Computer Society Press. To appear.

[18] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX - Workshop Proceedings - Micro-kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, April 1992.

[19] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[20] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

[21] S. Thibault, J. Marant, and G. Muller. Adapting distributed applications using extensible networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 234–243, Austin, Texas, May 1999. IEEE Computer Society Press.

[22] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation – application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May–June 1999.

[23] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. In *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 1993. ACM Operating Systems Reviews, 27(5), ACM Press.

[24] D.J. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 1999 ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.