

A DSL Approach to Improve Productivity and Safety in Device Drivers Development*

Laurent Réveillère[†] Fabrice Mérillon
Charles Consel[†] Renaud Marlet[‡] Gilles Muller

Compose Group, IRISA / INRIA, University of Rennes I
Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France

E-mail: {lreveill, merillon, consel, marlet, muller}@irisa.fr

Abstract

Although peripheral devices come out at a frantic pace and require fast releases of drivers, little progress has been made to improve the development of drivers. Too often, this development consists of decoding hardware intricacies, based on inaccurate documentation. Then, assembly-level operations need to be used to interact with the device. These low-level operations reduce the readability of the driver and prevent safety properties from being checked.

This paper presents an approach based on domain-specific languages to overcome these problems. We define a language, named Devil, dedicated to defining the basic communication with a device. Unlike a general-purpose language, Devil allows a description to be checked for consistency. This not only improves the safety of the interaction with the device but also uncovers bugs early in the development process.

To assess our approach, we have shown that Devil is expressive enough to specify a large number of devices. To evaluate productivity and safety improvement over traditional development in C, we report a first experiment based on mutation testing.

1. Introduction

Many appliances are now equipped with processors (*e.g.*, cellular phones, smart cards, cars, etc.) and many new peripheral devices are being developed. PC devices are also a rapidly evolving area. Typical examples are video adaptors, which come out at a frantic pace (every 6 months) to

*This work has been partly supported by Thomson Multimedia under the contract 199C031 and the French Ministry of Education and Research.

[†]Author's current address: LaBRI / ENSERB, 351 cours de la Libération, F-33405 Talence Cedex, France.

[‡]Author's current address: Trusted Logic, 5 rue du Bailliage, F-78000 Versailles, France. E-mail: Renaud.Marlet@trusted-logic.fr.

match the needs of ever demanding computer games. In such a competitive context, time-to-market is essential: device drivers need to be available as soon as a new device is ready.

A device driver is situated between the device and the operating system kernel (or directly in the application for small systems). It is a critical piece of code: miscommunication with either end may create major problems. On the one hand, the driver can incorrectly use the device and thus disable it. On the other hand, the device driver may misuse the resources made available by the OS (or application) and crash the system.

The preceding observations show that there is an acute need for both productivity and safety in device driver development. However, this need is difficult to satisfy for several reasons originating in the nature of the devices and their documentation, in the lack of adequate programming language support, and in the common programming practices.

An approach based on domain-specific languages

Yet, a good productivity should be possible to achieve as device drivers share a lot of commonalities and cover a very specific domain; this calls for some form of code re-use as well as expertise re-use. To that end, we propose a novel approach not only to achieve both kinds of re-use but also to systematize them. Our approach is based on domain-specific languages (DSLs) [7], as opposed to general-purpose languages (GPLs). Besides productivity, this approach also improves safety, without compromising efficiency.

A DSL is well adapted to this task because a DSL captures and abstracts the design and implementation expertise of a domain, the DSL programmer only has to focus on *what* to compute or to describe, as opposed to *how* to do it. In other words, fundamental concepts and issues that must be addressed by the programmer are made explicit in the DSL while implementation issues stay hidden; this improves pro-

ductivity. Moreover, because the expressive power of a DSL can be restricted, properties that are critical to the domain can be made decidable and checked automatically, thus improving safety. Besides, systematic domain-specific optimizations that are tedious and difficult to manually implement with a GPL can be automated and systematized in a DSL compiler.

Because device drivers include several domains of expertise (hardware and operating system issues), a good separation of concerns actually requires a separate DSL for each conceptual layer. This multi-DSL approach is further motivated by the fact that the different layers of a driver could be described by different programmers, with a specific background and specific constraints.

This paper

In this paper, we do not address the complete development of device drivers but focus on one of the conceptual layers. We introduce a DSL named Devil (for DEVICE Interface Language) that provides the low-level layer of a device driver, *i.e.*, the basic interaction with the device. We only consider *local devices*, *i.e.*, pieces of hardware that can directly communicate with the CPU using I/O, address and data buses. We do not consider *remote devices* such as printers, that actually communicate at a higher level through local devices such as serial or parallel interfaces.

A Devil specification rigorously describes the access mechanisms, the type and the layout of data that are exchanged to operate the device, as well as some behavioral properties. It does not assume any particular OS, and can therefore be used for any target platform. As a matter of fact, using the language only requires some hardware expertise; a description in Devil could typically be written by the device vendor.

Compiling a Devil description provides a typed, high-level interface to the device which can be used to write the upper layers of the device driver. For the driver programmer, the benefit of this approach is that the interface models an idealized device and abstracts the hardware intricacies. The upper layers can either be written in a GPL, which directly utilizes the implementation of the device interface, or in the future, combined with programs written in other DSLs for drivers. Because Devil is a restricted language, various typing and consistency properties of a specification can be verified. Because the generated interface is strongly typed, its use in the upper layers of the device driver can also be checked by a standard GPL compiler.

Our contributions are the following.

- We have carried out a domain analysis on device drivers, which points out the difficulties of driver development. By separating the concerns of hardware

vendors and OS driver programmers, we have extracted and structured the key concepts as well as the commonalities and variations in the code used for communicating with a device. We have also identified important properties that provide safety guarantees for such communication.

- Based on this analysis, we have designed a language (Devil) to precisely describe the interaction with hardware devices and to provide a high-level software interface for operating them. The language is strongly typed, and enables consistency properties to be checked on the specification as well as on the use of the corresponding interface in a driver. Such verification would be impossible to perform on drivers written using a GPL.
- To assess the usefulness of the language, we have shown that it is expressive enough to describe a wide range of standard PC devices including Ethernet, video, IDE disk, sound, interrupt, DMA and mouse controllers.
- To evaluate the productivity and safety improvement offered by Devil, we have conducted a mutation testing experiment. This evaluation demonstrates that a driver written in C but using the Devil-generated library may contain from 60% to 500% times fewer errors than an equivalent driver fully written in C.

The paper is organized as follows. Section 2 analyzes the difficulty of device driver development. Section 3 presents DSLs and argues why they are well suited for specifying device drivers. Section 4 describes the design of the Devil language. Section 5 presents the mutation testing experiment. Section 6 describes related work. Finally, Section 7 concludes and suggests future work.

2. Writing low-level device drivers

We have performed a domain analysis of device driver development. The following points summarize various reasons why developing device drivers is difficult.¹

Devices are complex. The design of a device is subject to numerous, sometimes contradictory constraints such as performance requirements and backward compatibility. As a result, the programming interface of a device is often awkward: contorted addressing modes, random partitioning of device registers, obscure initialization sequences, etc.

¹We concentrate here on the low-level part of the drivers, *i.e.*, the communication with the device; the higher-level layers raise other kinds of issues, such as proper management of OS resources.

Devices are inaccurately documented. Device drivers are based on the documentation made available to the programmers by the hardware vendor. This documentation is not easy to read since low-level and high-level concepts are generally intertwined: electronics, communication mechanisms, physical placement of data (register layout), semantics. The terminology used also changes from one device vendor to another. Moreover, any documentation still is informal because it is written in a natural language. Consequently, a device description is often ambiguous, incomplete, or even inconsistent. In fact, there is no systematic way for the hardware vendor to validate a device specification.

Mapping device documentation into code is not straightforward. Extracting the hardware interface from the documentation and expressing it in a program requires a significant work, that is also tedious and error-prone. On the one hand, the device specification is expressed in terms of ports, registers (that are possibly indexed or paged), bit vectors (register fragments and corresponding values), etc. On the other hand, manipulating a device in a driver requires the use of assembly-level operations: explicit I/O bus accesses and bit manipulation operators (shift, and, or, etc.).

Language primitives are not adequate. While programming languages have put programmers further and further away from the functional units of a CPU, they only offer low-level instructions to operate a device. These assembly-level operations account for a significant part of a device driver; they represent between 10% and 20% of the lines of code. Because of their low-level nature, such operations are not checked for type-correctness and other consistency properties. Moreover, they are fairly unreadable.

Language abstraction mechanisms are inappropriate and little used. A common approach to reducing the effects of low-level operations on programmability and readability is to introduce macros, as available in the C language. In practice, macros in existing drivers are mostly used to give symbolic names to specific constants, such as bit masks or I/O port offsets. Very few of the drivers we examined use macros to encapsulate a whole set of low-level operations. The reason is that, although the patterns of communication with the device are similar, there is little sharing from one device register to another. On the one hand, the abstraction of low-level code fragments into generic, reusable macros is considered too tedious. On the other hand, defining more specific macros, such as individual register accessors, is not considered useful as they would often only be used once or twice; even though doing so would make the resulting code more readable. In any case, the use of macros depends on the programmer's care and customs. Typically,

it is neither systematic nor uniform: many literal constants are hardwired in the code and definitions are often scattered into different source files without apparent reason.

Speed is an overrated concern. Although they offer better typing guarantees than macros and can be inlined for efficiency, functions are almost never used because they are reputed too slow. The fact is that speed is a crucial issue for many device drivers and that not all compilers support function inlining. Still, instead of focusing on critical paths, device driver programmers tend to always keep performance in mind and optimize every piece of code.

Programming practices are poor. The above issues show that some deficiencies in the software engineering of many device drivers today also originate in the attitude of programmers. In fact, people who write device drivers, especially for Linux, are proud to have written code that is incomprehensible, even to their peers.

Low-level device drivers require programmers with two domains of expertise. The unique situation of a device driver requires an expertise in two different areas. On the one hand, specific hardware expertise is needed to understand the low-level interface of the device and its internal behavior. On the other hand, software engineering expertise is required to impose a programming style, to structure the code (*e.g.*, by defining appropriate abstractions for the low-level parts), to achieve efficiency, yet to make the code open enough to enable future extensions. As a matter of fact, few driver programmers can be considered expert in both domains.

Example. Consider an excerpt of an actual driver (logitech mouse), displayed in Figure 1. As can be seen, the values of `dx`, `dy` (the horizontal and vertical motion of the mouse), and `buttons` are constructed using interleaved bit operations and device accesses. This code fragment is a compelling example of the awkward programs that can be written at a large scale when appropriate support for manipulating low-level device functionalities is not available and when good programming practices are not favored.

Assessing the software engineering of device drivers

The complexity of devices, the inaccuracy of the documentation, the complex mapping from a device specification to code, the unsuitability of language support and the programmers' practice and expertise all have a significant impact on the software engineering of device drivers. As a matter of fact, developing drivers is not an efficient process and often leads to code that is intricate and unreadable, which impedes maintenance and evolution. Not sur-

```

#define MSE_DATA_PORT          0x23c
#define MSE_CONTROL_PORT      0x23e
...
#define MSE_READ_X_LOW        0x80
#define MSE_READ_X_HIGH       0xa0
#define MSE_READ_Y_LOW        0xc0
#define MSE_READ_Y_HIGH       0xe0      1a. Definition

```

```

outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);
dx = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_X_HIGH, MSE_CONTROL_PORT);
dx |= (inb(MSE_DATA_PORT) & 0xf) << 4;
outb(MSE_READ_Y_LOW, MSE_CONTROL_PORT);
dy = (inb(MSE_DATA_PORT) & 0xf);
outb(MSE_READ_Y_HIGH, MSE_CONTROL_PORT);
buttons = inb(MSE_DATA_PORT);
dy |= (buttons & 0xf) << 4;
buttons = ((buttons >> 5) & 0x07);      1b. Use

```

Figure 1. Example of communication with the Logitech bus mouse (Linux 2.2.12)

prisingly, this situation can have a disastrous impact on the reliability of commercial operating systems. For example, Microsoft reports that 44% of system failure in NT4 are caused by drivers².

Yet, drivers development is a repetitive process and is built on patterns of code that are specific to the domain: bus transactions, bit manipulations, usage patterns of OS resources, fixed API, etc. This observation calls for re-use, to improve both productivity and safety. However, as mentioned above, general-purpose languages offer little support. First, code patterns are too fine-grained to be usefully abstracted. Second, typing rules, that are about the only validation mechanism offered by GPLs, are too loose to detect bugs early in the development process; the programmer can only rely on testing with sample data.

Domain-specific languages offer a solution to all these problems. Because they offer suitable built-in abstractions, they capture domain expertise and systematize re-use, regardless of the programmer's practice. Moreover, they provide additional safety guarantees as they allow domain-specific properties to be automatically checked. DSLs are further presented in the next section.

3. Domain-specific languages

A domain-specific language is a programming or specification language dedicated to a particular domain or problem. A DSL provides appropriate built-in abstractions and notations; it is often small, more declarative than imperative, and less expressive than a general-purpose language.

Examples of DSL are numerous. Some are distributed

²As from a sample from Product Support Services for NT-Server 4.0, May–July 1999, communication by Jim Allchin (Senior Vice President in charge of Windows 2000), COMDEX, November 15th 1999.

worldwide and used on a daily basis, e.g., SQL, Unix shells, makefiles, etc. DSLs have been used in various domains such as graphics [12, 15], financial products [2], telephone switching systems [13, 17], protocols [5, 23], operating systems [20], device drivers [25], routers in networks [23] and robot languages [3]. This profusion shows the recent attention that DSLs have received from both the research and industrial communities.

The following points explain why DSLs are more attractive than GPLs for a variety of applications.

Easier programming. Because of appropriate abstractions, notations and declarative formulations, a DSL program is more concise and readable than its GPL counterpart. Hence, development time is shortened and maintenance is improved. As programming focuses on what to compute as opposed to how to compute, the user does not have to be a skilled programmer. Specific optimization strategies can be implemented in the DSL compiler not only to offer performance but also to systematize it.

Systematic re-use. Most GPL programming environments include the ability to abstract common operations into libraries. However, re-use of libraries is left to the programmer. In contrast, a DSL offers guidelines and built-in functionalities which enforce re-use. Additionally, a DSL captures domain expertise, either implicitly by hiding common program patterns in the DSL implementation, or explicitly by exposing appropriate parameterization to the DSL programmer. Thus, the programmer necessarily re-uses library components and domain expertise.

Improved safety. DSLs enable more properties about programs to be automatically checked. In contrast to a GPL, the semantics of a DSL can be restricted to make decidable some properties that are critical to a domain [23]. Detecting errors early in the development process also improves productivity. In addition, as re-use is not only improved but systematized, DSL programs rely on components that are frequently used and thus well tested.

DSLs as a software architecture

Not all applications call for a DSL. In fact, a DSL only makes sense to structure and implement a program family. A *program family* is a set of programs that share enough characteristics that it is worthwhile to study and develop them as a whole [18]. A program family is typically associated to a given problem in a given domain. A DSL program can be viewed as a way to designate a member of a program family. A DSL compiler then acts as an application generator which can produce any member of the program family.

In practice, the formulation in terms of a DSL suggests an attractive way to architecture software to implement a program family [7]. A program family is traditionally implemented using a library that captures common code patterns and offers re-use for implementing the various family members. As a library becomes larger or more generic, its usability decreases due to the multiplication of entry points, parameters and options offered. As a result, the library might be ignored by programmers because it is considered too complex to use or too difficult to read. In this situation, a DSL can offer a domain-specific interface to the library so that the programmer does not have to directly manipulate numerous highly-parameterized building blocks; the complexity is hidden. To that effect, the DSL compiler automatically generates code that calls the library functions; the library can then be seen as an abstract machine for the DSL. The generated code corresponds to the code that would be manually written to implement a family member using the library.

Designing and developing a DSL

The definition of a DSL critically relies on a thorough analysis of the commonalities and variations in a program family, which identifies common patterns in the design and implementation [7]. The goal of this analysis is also to extract the key abstractions, properties, notations and terminology used in the domain. It contributes to determining the basic elements of the language to be designed, as well as possible or required verifications to be performed on programs.

DSLs for device driver development

Device drivers form a good example of a tight program family: in addition to having the same API (for a given operating system and type of device), they all share similar operations, although they vary according to the hardware. They are thus a good target for DSLs. Moreover, as shown in Section 2, the productivity and safety of device driver development are poor. These are software engineering concerns that are addressed by DSLs. Additionally, DSLs can also address the efficiency issue, which can be a major concern in device driver development.

4. Design of Devil

This section gives a summary of our domain analysis of the lower layer of drivers (*i.e.* the communication with devices). For each identified concept, we present the corresponding Devil language construct.

4.1. Domain analysis

To perform our domain analysis, we exploited a variety of information sources. We thoroughly examined a wide spectrum of devices and their corresponding drivers, mainly from Linux sources: Ethernet, video, sound, disk, LED display, interrupt, DMA and mouse controllers. This study was supported by literature about driver development [8, 21], device documentations available on the web, and discussions with device driver experts for Windows, Linux and embedded operating systems.

The overall result of our domain analysis shows that a language is needed to provide a high-level software interface to hardware devices. As is usually the case for interfaces [10, 11], our language should have a declarative nature.

4.2. Levels of abstractions

The top level of a Devil specification is the declaration of a device. Physical addresses, abstracted as *ports* or ranges of ports, parameterize the declaration. Ports then allow device *registers* to be declared. Finally, *device variables* are defined from registers, forming the functional interface to the device. These three levels of abstraction are illustrated by a simplified fragment of the Devil description of a mouse controller, displayed below.

```
device logitech_busmouse(base : bit[8] port@{0..3})
{
  register sig_reg = base@1;
  variable signature = sig_reg : int(8);
  ...
}
```

The top-level declaration introduces the `logitech_busmouse` description. This description is parameterized with respect to a range of ports provided as the main address base and a range of offsets (from 0 to 3). An eight-bit register `sig_reg` is declared at port `base`, offset by 1. Finally, the device variable `signature` is the interpretation of this register as an eight-bit unsigned (by default) integer. The resulting description fragment declares a device whose functional interface consists of a single device variable (`signature`). Only device variables are visible from outside a Devil description (unless they are declared private); ports and registers are hidden since these abstractions are not part of the functional interface of the device.

Let us now examine in detail each of these levels.

4.3. Ports

The port abstraction is at the basis of the communication with the device. This abstraction hides the fact that, depending on how the device is mapped, it can be operated via I/O

and memory operations. Since a device often has several communication points whose addresses are derived from a few main addresses, Devil includes a port constructor, denoted by @, which takes as arguments a ranged port and a constant offset (e.g., base@1 as illustrated above). To limit the set of accessible ports to those that are meaningful for the given device, the range of valid offsets must be specified (e.g., port@{0..3} as illustrated above).

4.4. Registers

Based on our domain analysis, registers are typically defined given two ports: a port for reading and a port for writing. Only one port needs to be provided when reading and writing share the same port (as is the case for sig_reg, shown above), or when the register is read-only or write-only (see example below). Registers also have a size (number of bits), which must be explicitly specified.

Bit masks. A register declaration may be associated with a mask to specify the constraints on bits of this register. Each symbol in the mask corresponds to a bit in the register. A symbol can either be '.' to denote a relevant bit, '0' or '1' for an irrelevant bit but with a fixed value (0 or 1) when read or written³, or '*' for an irrelevant bit whether read or written. By default, if no mask is specified, all bits of a register are assumed relevant. As an example, consider the declaration of the register below.

```
register index_reg = write base@2, mask '1..00000';
```

The mask indicates that only bits 6 and 5 are relevant⁴. Bit 7 must have value 1 when written. Similarly, bits 4 through 0 must have value 0 when written. Only the relevant bits of a register can be used to construct a variable. In the example below, the two relevant bits make up a two-bit unsigned integer variable (i.e., a variable that can take values from 0 to 3).

```
private variable index = index_reg[6..5] : int(2);
```

Access pre-actions. Some registers require other registers to be set to specific values before being accessed. For example, indexed registers can be viewed as a sequence of registers with a fixed base address; accessing such registers typically consists of manipulating two ports: one to set the index of the register to be accessed and one to read or write the target register. To do so, pre-actions may be attached to a register to set up a specific context before it is read or written. The following example declares two read-only

³This can be a hardware constraint or a provision made by the device vendor to allow future extensions.

⁴Following the convention used in device and chip documentation, bits are numbered from right to left, starting with 0.

registers that can be accessed at the same port base@0, provided that the device variable index is set either to 0 or 1.

```
register dx_low =  
  read base@0, mask '****...', pre {index = 0};  
register dx_high =  
  read base@0, mask '****...', pre {index = 1};
```

4.5. Device variables

For hardware efficiency reasons (e.g., to minimize the number of I/Os), a register may group various independent values. For example (see Figure 1), three bits in a register may be used to denote which buttons of a mouse have been pushed, while the remaining bits of the register may provide information concerning the motion of the mouse. In other cases, some meaningful values have to be constructed by assembling bit sequences from different registers. For example, the mouse motion dx in Figure 1 is encoded in the device using the lowest four bits of two different registers. In fact, those meaningful values, that are possibly spread over several register fragments, represent a convenient way to express high-level communications with the device. As they can conceptually be read or written like any variable in a GPL, we call them *device variables*. Since they do not directly map to physical entries, these variables correspond to a logical view of the device; they abstract over the physical representation of the device state. In essence, device variables form the *functional interface of the device* to be used by the programmer.

Construction of values. Previous examples of device variables have shown declarations corresponding to an entire register (signature) and register fragments (index). It is also possible to declare a variable as a combination of these, as illustrated in the following example.

```
variable dx =  
  dx_high[3..0] # dx_low[3..0] : signed int(8);  
variable dy =  
  dy_high[3..0] # dy_low[3..0] : signed int(8);
```

The horizontal and vertical motion of the mouse is constructed by the concatenation (using the # operator) of the two fragments of the motion registers that store the low and high four bits of the actual motion values. The resulting eight-bit sequences are interpreted as signed integers.

Types. Devil allows bit sequences to be interpreted as a given type. The set of types currently offered by Devil reflects the types used in the various device drivers that were studied during the domain analysis: booleans, signed or unsigned integers of various sizes, enumerated types, ranges or sets of integers. For lack of space, examples of these constructs are omitted.

Some other features of Devil are not detailed here. These features include access post-actions, enumerated types and arrays, structures to synchronize device variables, order of register accesses, register constructors, variable behaviors and conditional declarations depending on device modes. A detailed description of Devil can be found in [22].

4.6. Verification

In contrast with GPLs, a DSL makes domain-specific information explicit. In Devil, declarations enable three categories of verification that are beyond the scope of GPLs. First, because Devil is strongly typed, all uses of the entities (*e.g.*, ports, registers, variables) can be matched against their types. Second, omitted or double definitions can be detected. For example, all bits of registers can be checked as being used at least once⁵: no device variable definition is thus missing. Lastly, overlapping definitions, *i.e.*, building entities used more than once, can be located and reported as an error. For example, the same register bits cannot be included in two different variables.

The use of Devil's functional interface in a GPL also provides opportunities for verifications. These verifications include type checking and conditional variable checking; they can be both static and dynamic [22].

5. Assessment

In the previous sections, we have shown that DSLs enable more properties about programs to be automatically checked. In this section, we examine the number of errors detected (*i.e.*, covered) by both a GPL and a DSL. The GPL used in our study is the C language, since it is traditionally used to write device drivers. Using Devil as an alternative to GPLs introduces two languages in the development process: Devil for specifying a device interface and C (this use of C is denoted by C_{Devil} in the rest of the paper) for using the library that implements the Devil description. This approach leads to an evaluation of the pair $C_{\text{Devil}}+\text{Devil}$, and thus of C_{Devil} and Devil.

The evaluation of the *error-detection coverage* of Devil and C is based on mutation analysis. This evaluation enables us to demonstrate the benefits of the DSL approach in terms of software robustness.

DeMillo and Mathur have analyzed [9] the errors of $\text{T}_{\text{E}}\text{X}$ reported by Knuth [16]. Their analysis clearly reveals that simple errors do represent a significant fraction, though not the majority, of the errors in production programs. It also reveals that such errors remain hidden for a long time before testing exposes them. These observations are even more important considering the permissive nature of a language such as C.

⁵Some bits can be declared as irrelevant using bit masks though.

For a program P , mutation testing produces a set of alternate programs. Each alternate program, P_i , known as a *mutant* of P , is formed by modifying one statement of P at a time, according to some predefined mutation rules. These mutation rules are derived empirically from studies of errors commonly made by programmers when translating requirements into code [1].

In traditional mutation testing, we want to reason about the coverage of a set t of tests with respect to a program P . Mutation testing works on the principle that if t adequately covers P , then some test in t should be able to discriminate P from a mutation P' . Presumably, if a mutation cannot be discriminated by some test in t , then t does not adequately cover P . The proportion of mutants that *die* during mutation testing indicates how well P is covered by t .

A compiler (in our case, a C compiler or a Devil compiler) can be thought of as a test set, if we consider each analysis performed by the compiler to be a test (*i.e.*, an element of t). We'll say that the compiler adequately covers a program P if some analysis can discriminate P from every mutation P' .

Our study focuses on three different devices (Busmouse, Ethernet card, and IDE controller) and their corresponding C drivers⁶. Our experiment consists of measuring the error-detection coverage of C_{Devil} , Devil and C as discovered by the corresponding compilers/checkers⁷. Mutation rules are defined so as to ensure that the resulting mutant is syntactically correct, and actually modifies the semantics of the program. Mutation rules for C and Devil are always chosen as to favor C. Our experiment thus reflects worst cases for Devil.

Mutants are generated at a given program point. Such a program point is called a site. Each site leads to several mutants. For example, given an integer of two digits in base 16, 80 mutants can be generated (2 for removing a digit, 48 for inserting a new digit, and 30 for replacing a digit). Table 1 shows, for each target device in our experiment, the number of sites where mutants are generated.

Device		C	Devil	C_{Devil}	$\text{Devil}+C_{\text{Devil}}$
Logitech Busmouse	Lines of code	36	21	18	39
	Mutation sites	62	81	21	102
IDE (Intel PIIX4)	Lines of code	64	127	81	208
	Mutation sites	95	277	42	319
Ethernet (NE2000)	Lines of code	204	144	137	281
	Mutation sites	247	456	258	714

Table 1. Number of mutation sites

Coverage Analysis. Figures 2 and 3 summarize the results of our mutation analysis performed on C, Devil, and

⁶C drivers come from the Linux kernel version 2.2.12.

⁷Beside static verification, Devil also provides dynamic checking. This dynamic checking is not taken into account in our experiment.

C_{Devil} . The x-axis consists of the device drivers used in the study. The y-axis of Figure 2 represents the rate of undetected mutants per site. The y-axis of Figure 3 represents the number of undetected mutants balanced by the number of sites.

These analysis data demonstrate that the propensity of introducing undetected errors is 60% to 500% times lower when using Devil rather than using C only. It can also be observed that errors in the Devil part of the driver are nearly always detected.

These results can be further improved as the specifications and the compiler used in the experiment rely on an earlier version of Devil which does not exploit all of its features.

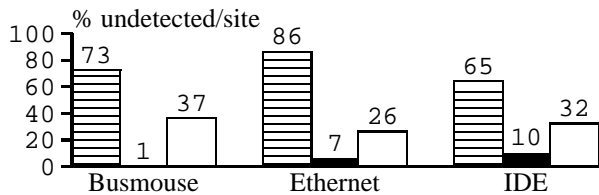


Figure 2. Percentage of undetected errors/site

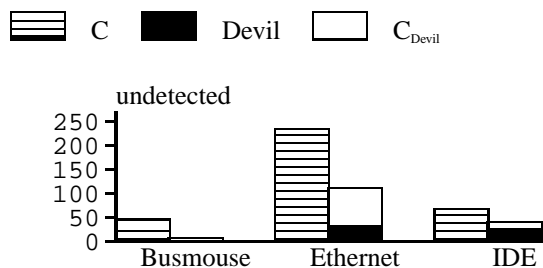


Figure 3. Number of undetected errors

6. Related work

Our work on device drivers started with a study of graphic display adaptors for a X11 server. We developed a language, called GAL, aimed at specifying device drivers in this context [25]. Although successful as a proof of concept, GAL covered a very restricted domain. It is this restriction which allowed us to model the domain with a single DSL.

The UDI project⁸ aimed at making device drivers source-portable across OS platforms. To do so, it normalizes the API between the OS and the lower part of device drivers [19]. This interface is being implemented as a library. Besides showing the timeliness of our work, UDI is complementary to Devil. Furthermore, UDI is a good basis for the development of our future DSLs for the upper layers

⁸The UDI (Uniform Driver Interface) project is the result of a contribution of several computer companies including Compaq, HP and IBM.

of a driver since this library already identifies the fundamental operations of these layers.

Windows-specific driver generators like BlueWater System's WinDK [4] and NuMega's DriverWorks [6] offer GUIs aimed at specifying the main features of the driver to be generated. They produce a driver skeleton that consists of invocations of coarse-grained library functions. To our knowledge, no existing driver generators cover the communication with the device.

Languages for specifying digital circuits and systems have existed for a while. A standard language, widely used in this domain, is VHDL [14]. A VHDL specification describes the low-level logic and electronic functionalities of a device. Devil differs from VHDL in that it concentrates on communication with the device, not the device's inner workings. The interface described by a Devil specification can not be deduced by a VHDL specification.

7. Conclusion

Although devices are rapidly evolving and require fast releases of drivers, driver development has received little attention from the research community. This situation is surprising when considering the level of safety that drivers should offer to guarantee the integrity of their host system.

In this paper, we have presented the following results. We have analyzed the domain of low-level device drivers and listed obstacles to fast production of safe drivers. We have pointed out that device drivers form a program family that could be described using domain-specific languages. Based on our domain analysis, we have designed a language, named Devil, aimed at specifying the communication layer with a device, providing a typed, functional interface. Besides strong typing, this language allows the consistency of domain-specific properties to be automatically checked. Errors are thus detected early in the development process and safety is improved. This approach sharply contrasts with the use of a general-purpose language which requires writing tedious and error-prone assembly-level code and which does not permit any useful validation.

To assess our approach, we have shown that Devil is expressive enough to specify the interface of a large spectrum of different PC devices including Ethernet, video, disk, sound, interrupt, DMA and mouse controllers. Implementations generated from such specifications have not shown any significant performance loss. To evaluate the effectiveness of strong typing and consistency checking, we have performed a mutation testing experiment, comparing the static error detection of both approaches.

Devil improves productivity by providing domain-specific abstractions that contribute to make programming easier. Increased productivity also comes from reducing undetected errors, as illustrated by our mutation analysis when

using Devil rather than C alone. More generally, a specification written in Devil improves productivity by abstracting the device interface in an OS independent way, allowing systematic reuse.

Following our approach to driver development, our future work aims at designing DSLs to model the upper layers of drivers. These DSLs will make it possible to verify drivers from the device interface to the operating system.

Availability. Devil specifications as well as an implementation of a compiler/checker are available from <http://www.irisa.fr/compose/devil>.

Acknowledgment. The authors thank Anne-Françoise Le Meur and Julia Lawall for their comments on earlier versions of this paper, and Robin Hansen who wrote part of the Devil compiler.

References

- [1] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Centre, Purdue University, West Lafayette, Indiana, Mar. 1989.
- [2] B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language describing financial products. In *IEEE Workshop on Formal Methods Application in Software Engineering*, pages 6–13, Apr. 1995.
- [3] E. Bjarnason. Applab: a laboratory for application languages. In L. Bendix, K. Nørmark, and K. Østerby, editors, *Nordic Workshop on Programming Environment Research, Aalborg*. Technical Report R-96-2019, Aalborg University, May 1996.
- [4] BlueWater Systems, Inc. *WinDK Users Manual*. URL: www.bluewatersystems.com.
- [5] S. Chandra and J. Larus. Experience with a language for writing coherence protocols. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, Oct. 1997.
- [6] Compuware NuMega. *DriverWorks User's Guide*. URL: www.numega.com.
- [7] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in Lecture Notes in Computer Science, pages 170–194, Pisa, Italy, Sept. 1998.
- [8] E. N. Dekker and J. M. Newcomer. *Developing Windows NT device drivers : A programmer's handbook*. Addison-Wesley, first edition, Mar. 1999.
- [9] R. A. Demillo and A. P. Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical Report SERC-TR-92-P, Software Engineering Research Centre, Purdue University, West Lafayette, Indiana, Feb. 1991.
- [10] R. Draves, M. Jones, and M. Thompson. *MIG - The MACH Interface Generator*. School of Computer Science, Carnegie Mellon University, July 1989.
- [11] 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.
- [12] C. Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, Oct. 1997.
- [13] N. Gupta, L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss. Auditdraw: Generating audits the fast way. In *Proceedings of the Third IEEE Symposium on Requirements Engineering*, pages 188–197, Jan. 1997.
- [14] IEEE Standards. *1076-1993 Standard VHDL Language Reference Manual*, 1994. URL: standards.ieee.org.
- [15] S. Kamin and D. Hyatt. A special-purpose language for picture-drawing. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, Oct. 1997.
- [16] D. E. Knuth. The errors of \TeX . *Software Practice and Experience*, 19(7):607–685, July 1989.
- [17] D. Ladd and C. Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, New Mexico, Oct. 1994.
- [18] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9, mar 1976.
- [19] Project UDI. *UDI Specifications, Version 1.0*, September 1999. URL: www.project-udi.org.
- [20] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for operating system specialization. In *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, Jan. 1997. Computer Science Technical Report, University of Illinois at Urbana-Champaign.
- [21] A. Rubini. *Linux Device Drivers*. O'Reilly, first edition, Feb. 1998.
- [22] L. Réveillère, F. Méryllon, C. Consel, R. Marlet, and G. Muller. The Devil language. Research Report 1319, IRISA, Rennes, France, May 2000.
- [23] 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, Oct. 1998.
- [24] S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device driver: from design to implementation. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, Oct. 1997.
- [25] 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. Extended version of [24].