HAL
archives-ouvertes.fr

Towards Verifiable Device Drivers: An Approach based on Domain-Specific Languages

Fabrice Mérillon, Laurent Réveillère, Charles Consel, Robin Hansen, Renaud Marlet, Gilles Muller

▶ **To cite this version:**

Fabrice Mérillon, Laurent Réveillère, Charles Consel, Robin Hansen, Renaud Marlet, et al.. Towards Verifiable Device Drivers: An Approach based on Domain-Specific Languages. [Research Report] RR-3809, INRIA. 1999. inria-00072849

HAL Id: inria-00072849
https://hal.inria.fr/inria-00072849

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Verifiable Device Drivers: An Approach based on Domain-Specific Languages

Fabrice Mérillon Laurent Réveillère

Charles Consel Robin Hansen Renaud Marlet Gilles Muller

N°3809

Novembre 1999

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--3809--FR+ENG

INRIA
RENNES

Towards Verifiable Device Drivers: An Approach based on Domain-Specific Languages

Fabrice Mérillon Laurent Réveillère
Charles Consel Robin Hansen Renaud Marlet Gilles Muller

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n ° 3809 — Novembre 1999 — 14 pages

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 ambiguous or incomplete documentation, to determine how to operate a device. Then, assembly-level operations need to be used to interact with the device. These low-level operations make the device driver fairly unreadable and prevent safety properties from being checked.

This paper presents a language, named Devil, dedicated to defining the functional interface of a device. More precisely, Devil aims at specifying the access mechanisms, the type and layout of data, and behavioral properties involved in operating a device.

The benefit of our approach is that, once compiled, a Devil description implements an interface which models an idealized device and abstracts the hardware intricacies. Unlike a general-purpose language, Devil allows a description to be thoroughly verified; this verification greatly improves the safety of the communications with the device. The design of Devil is based on key concepts we identified in analyzing the domain of device drivers.

Our language has been used to specify a large number of PC devices including Ethernet, video, sound, interrupt, DMA and mouse controllers.

Key-words: device drivers, DSLs

(Résumé : tsvp)

This research is supported in part by Thomson Multimedia under the contract 199C031.

Vers des pilotes de périphérique vérifiables : une approche fondée sur les langages dédiés

Résumé : Pour suivre la cadence effrénée à laquelle les périphériques sortent sur le marché, les pilotes de périphérique doivent être produits de plus en plus rapidement. Pourtant, peu de progrès ont été faits pour améliorer leur développement. Pour écrire un pilote, il faut trop souvent commencer par comprendre le fonctionnement du périphérique, en déchiffrant tant bien que mal une documentation ambiguë ou incomplète. Ensuite, pour interagir avec le périphérique, il faut recourir à un style de programmation proche de l'assembleur. Ces opérations bas-niveau rendent le pilote assez illisible et ne permettent pas de vérifier des propriétés de sûreté.

Cet article présente un langage nommé Devil, dont le but est de définir l'interface fonctionnelle d'un périphérique. Plus précisément, Devil vise à spécifier les mécanismes d'accès aux données, leur type et leur disposition, ainsi que des propriétés comportementales reliant ces données au fonctionnement du périphérique.

L'avantage de notre approche est qu'une fois que la description Devil d'un périphérique a été compilée, la complexité du périphérique disparaît derrière une interface abstraite et idéalisée. Contrairement aux opérations bas-niveau d'un langage généraliste comme C, une description écrite en Devil peut être vérifiée dans le détail. Cette vérification apporte un gain de sûreté significatif dans les interactions avec le périphérique.

Les concepts clés sur lesquels est fondée la conception de Devil ont été identifiés grâce à une analyse du domaine des pilotes de périphérique. Nous avons utilisé Devil pour spécifier de nombreux périphériques du monde PC, tels que des contrôleurs Ethernet, DMA, vidéo, son, de souris et d'interruptions.

Mots-clé : pilotes de périphérique, langages dédiés

1 Introduction

Many appliances are now equipped with processors (*e.g.,* cellular phones, smart cards, cars, ...) 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 match the needs of ever demanding computer games. In such a competitive context, time-to-market is essential: as soon as a new device is ready, drivers need to be rapidly available to enable applications to exploit it.

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 one hand, the driver can incorrectly use the device and disable it. On the other hand, the device driver may misuse the resources made available by the OS and crash the system.

There is thus an acute need for both productivity and safety in device driver development. However, this need is difficult to satisfy, for several fundamental reasons[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 awkard: contorted addressing modes, random partitioning of device registers, obscure initialization sequences, ...

Devices are incompletely documented. Typically, an informal description of the device (written in a natural language) is the only documentation available to the programmers writing the device driver. The low-level and high-level concepts are generally intertwined: communication mechanisms, physical placement of data (register layout), semantics. Moreover, the documentation may use a terminology that changes from one device vendor to another. Consequently, a device description is often ambiguous, inconsistent, incomplete and erroneous. In fact, there is no systematic way for the hardware designer to validate a description.

Devices are interfaced by low-level operations. While programming languages have put programmers further and further away from the functional units of a CPU, manipulating a device still requires the use of assembly-level operations. Typical computations consist of explicit bus accesses and bit manipulations. These assembly-level operations account for a large part of a device driver; once compiled, they represent between 16% and 30% of its assembly code (see Appendix A). Because of their low-level nature, such operations are not checked for type-correctness and other consistency properties. Moreover, they are fairly unreadable.

A common approach to limiting the effects of low-level operations on code readability is to introduce macros, as available in the C language. In practice, macros in drivers are mostly used to give symbolic names to specific constants. Very few of the drivers we examined use macros to encapsulate a whole set of low-level operations. Besides, the macro facility is far from being systematically used throughout the code of a device driver.

Consider an excerpt of an actual driver[2], displayed in Figure 1. As can be seen, the value `dx` (the horizontal motion of the mouse) is constructed using interleaved bit operations and device accesses. This code fragment is a compelling example of the awkward programs which can be written at a large scale when appropriate support for manipulating low-level device functionalities is not available.

Device drivers require programmers with three domains of expertise. The unique situation of a device driver requires an expertise in three different areas. Specific hardware expertise

[1] In the following, we focus on *local devices*, 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.

[2] Logitech bus mouse driver as available in Linux version 2.2.5.

```
#define MSE_DATA_PORT           0x23c
#define MSE_CONTROL_PORT        0x23e
#define MSE_READ_X_LOW          0x80
#define MSE_READ_X_HIGH         0xa0
```

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;
```

1b. Use

Figure 1: Communicating with a device in a typical driver[2]

is needed to understand the low-level interface of the device and its internal behavior. Software engineering expertise is required to impose a programming style, to structure the code (*e.g.,* by defining abstractions for the low-level parts), to achieve efficiency, yet to make the code open enough to enable future extensions. Finally, OS expertise is also needed to implement the required driver's API (Application Programming Interface), and to make a careful use of OS resources (*e.g.,* memory, synchronization, interrupts). As a matter of fact, few driver programmers can be considered expert in these three domains.

As a result of the above problems, the code of device drivers is often unreadable and costly to develop, maintain and evolve. Not surprisingly, this situation may have a disastrous impact on the reliability of commercial operating systems.

This negative assessment of the situation is difficult to understand considering the repetitive nature of the development of device drivers. In fact, inspection of existing device drivers shows that they share enough characteristics to be studied and developed as a whole. Examples include bus transactions, bit manipulations, usage patterns of OS resources, fixed API, ... Such programs are said to form a *program family* [8].

Because device drivers form a program family, their development could be systematized to improve productivity [3]. Because device drivers are developed at a fast pace, their development should systematize both expertise and code re-use to better time-to-market delivery [12, 9, 4]. Because device drivers are critical pieces of a system, their implementation should be verified to improve safety [3].

Our approach

Our approach to systematizing the development of device drivers is based on domain-specific languages (DSL) [3]. Because device drivers include several domains of expertise, good separation of concerns between hardware and software aspects actually requires one DSL for each conceptual layer. This multi-DSL approach is further motivated by the fact that each layer of a driver may be described by a specific programmer, with a specific background and specific constraints.

The design of each language is fueled by a thorough analysis of the domain of each conceptual layer (device documentations, existing code, ...). Each language factorizes commonalities and allows variations to be expressed by specific notations and abstractions. The domain analysis also enables critical safety properties to be identified. The semantics of each DSL can be suitably restricted to allow these properties to be verified, while preserving the expressiveness of the language [13]. Another goal of the design of our DSLs is to lead to compilation strategies which ensure no performance penalty compared to existing drivers.

This paper

We introduce in this paper a DSL named Devil[3] aimed at providing the lower layer of a device driver, *i.e.,* the basic interaction with the device. A Devil specification rigorously describes the

[3]Devil stands for DEVice Interface Language.

```
register index_reg = write base@2, mask '1**00000';
variable index = index_reg[6..5] : int(2);

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

variable dx = dx_high[3..0] # dx_low[3..0] : signed int(8);          2a. Definition
```

```
dx = get_dx();                                                       2b. Use
```

Figure 2: Devil definition corresponding to the C code in Figure 1

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 this language only requires hardware expertise; a description in Devil could typically be written by the device vendor.

Compiling a Devil description provides a typed, high-level interface of 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 general-purpose language (GPL), which directly utilizes the implementation of the device interface, or in the future, combined with programs written in other DSLs for drivers.

Figure 2 illustrates the benefits of Devil in terms of separation of concerns and readability; this example is detailed in the rest of the paper. As can be seen, all the communications with the device are encapsulated in an access function generated from the Devil description. Then, the driver programmer only has to focus on operating the device using abstract values. Because Devil is a restricted language, various typing and consistency properties of a specification can be verified. Because the generated interface is 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 the program family of device drivers. Separating the concerns of hardware vendors and OS driver programmers, we have extracted and structured the key concepts of the domain as well as the commonalities and variations in the design and development of code for communicating with hardware. We have also identified important properties that could improve the safety of those communications.
- Based on this analysis, we have designed a language to describe the interactions with hardware devices and to provide a high-level software interface for operating them. This 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. These verifications would be impossible to perform on drivers written with GPLs.
- We have shown that the language is expressive enough to describe a wide range of standard PC devices including Ethernet, video, sound, interrupt, DMA and mouse controllers. Examples of such descriptions are available online.

The paper is organized as follows. Section 2 gives an overview of the domain analysis. Section 3 describes how the language design is extracted from this analysis. Section 4 shows how the language restrictions are exploited to make important safety properties decidable. Section 5 gives the current implementation status of Devil. Section 6 compares the Devil approach to related works. Section 7 provides an assessment of Devil and discusses future work.

2 Domain Analysis

The design of a DSL critically relies on a thorough analysis of the commonalities and variations in a program family, as well as the fundamental concepts of the domain. The output of this analysis contributes to determining the basic elements of the language to be designed, as well as possible or required verifications to be performed on programs. In this section, we present the domain analysis of the lower layer of device drivers. The salient features of the Devil language and the verifications that it enables are presented in the following two sections.

Our goal in analyzing the domain is to extract the key abstractions, notations and terminology which underlie the basic interaction with a device. This phase also includes the study of common patterns in the design and implementation of drivers. We exploited a variety of information sources to perform this domain analysis. We thoroughly examined a wide spectrum of devices and their corresponding drivers, mainly from Linux sources: Ethernet, video, sound, interrupt, DMA and mouse controllers. This study was supported by literature about driver development [11, 5], device documentations available on the web, and device driver experts for Windows, Linux and embedded operating systems. Several key concepts came out of this work, that are summarized in the rest of this section.

Ports. Interacting with a device relies on the notion of communication points, often named *ports*. A port is used to read/write values from/to the device. These operations are implemented differently depending on how the device is visible by the processor (*e.g.,* mapped in memory or I/O). Most devices require several ports for communication; a few ports are often used as base addresses to define *ranges of ports*.

Registers. *Registers* represent the atomic units of information that can be exchanged with a device. These values are typically eight, sixteen, or thirty-two-bit words. Some registers may only be read and some only be written; others may be read or written. A single register may be accessed for reading at a given port and for writing at another port. A register may also require a specific *access path* involving any number of other registers (*e.g.,* indexed or paged registers). Even though access paths can be factorized with macros, examination of existing drivers shows that this facility is rarely used causing the driver code to be rather tangled.

Because of hardware considerations, some bits of a register can be reserved or unused, some must have a fixed value when written (or have a fixed value when read), and others correspond to values that are irrelevant. Bit constraints are usually well-described in device documentations in terms of *bit masks*. In a GPL like C, bit masks are translated into constants and bit operations for encoding or decoding a register value.

Device variables. For hardware efficiency reasons (*e.g.,* to minimize the number of I/Os), a register may group various independent values. For example, 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 and 2 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.

The description of the functional view of a device is typically spread over the whole device documentation. Each device variable is introduced when register fragments are explained. The description of a device variable includes its *type*, its *usage constraints* and its *behavior*. The types

of device variables are usually integers of various sizes, booleans, or bit patterns. Usage constraints express the fact that some variables can only be read or written. Behaviors address how the value of a variable evolves and how it impacts the device. Specifically, this value may be *stable* (*i.e.,* only updated by the driver) or be *volatile* (*i.e.,* updated by the device itself). Furthermore, a device variable may be *passive* (*i.e.,* be considered as a regular memory cell), or it may *trigger* device actions when written (*i.e.,* be a device command).

A device's documentation describes how device variables are constructed from registers, their types, usage constraints and behaviors. However, there is no systematic and rigorous way for a programmer to express this information in the driver code, and GPLs provide little or no support to facilitate this process. Information provided by a device documentation is thus often ignored and lost. Consequently, no verification is included in a device driver whether for ports, registers or device variables, as confirmed by examination of existing code.

Our domain analysis, outlined in this section, forms the basis for the design of the Devil language.

3 Language Design

This section gives a tour of the Devil language illustrated with fragments of actual device descriptions. The features of Devil address the concepts identified in the domain analysis. The language is declarative and allows a structured definition of the communication with a device.

3.1 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 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.

3.2 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. The use of the port constructor is illustrated at the second line of the Devil description shown 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. This feature is illustrated at the first line of the previous example.

3.3 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 can be explicitly specified or which defaults to the size of values that can be exchanged at the corresponding port.

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 when read but with a fixed value (`0` or `1`) when written[4], 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 relevant[5]. 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 paths. 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 registers that can be accessed at the same port `base`, 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};
```

Register constructors. Because many register definitions often follow a similar pattern (as in the preceding example), Devil offers a parameterized register constructor. This feature is illustrated in the example below.

```
register reg(i : int(2)) = read base@0, mask '----****', pre {index = i};
register dx_low  = reg(0);
register dx_high = reg(1);
register dy_low  = reg(2);
register dy_high = reg(3);
```

This fragment of the mouse description declares a register constructor `reg` that includes a pre-action aimed at setting the index of the register to be accessed. The parameter of the register constructor `reg` is typed; it must match the type of the variable `index`. This register constructor is then used to declare four indexed registers. As illustrated by this example, register constructors allow the declaration attributes of similar registers to be factorized.

3.4 Device Variables

As suggested by the domain analysis, declarations of device variables address how their values are constructed from registers, their types, usage constraints and behaviors. We will discuss each of these issues in turn.

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

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

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 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.

Behaviors. As found in the domain analysis, device variables may be either *stable* or *volatile*, and either *passive* or *trigger*. Devil provides variable qualifiers to specify such behavior. When read, a variable is stable by default but may be specified as being `volatile`. When either read or written, a variable is passive by default but may be specified as being `trigger`. Further refinements are needed for the case when a variable is written. Indeed, some variables only trigger actions when the written value fulfills a condition: being different from the value previously written, or belonging to a subset of the writeable values. Some of these various behaviors are exemplified in the following Devil fragment extracted from an Ethernet controller description (types have been omitted) [7].

```
register cmd = base@0;
variable st  = cmd[1..0], trigger write except NEUTRAL;
variable txp = cmd[2],    trigger write except NOP;
variable rd  = cmd[5..3], trigger write except NODMA;
private variable page = cmd[7..6] : int(2);
```

The above fragment includes four variables based on the register `cmd`. Three of these variables (`st`, `txp` and `rd`) trigger an action when written, except for specific values[6] (`NEUTRAL`, `NOP` and `NODMA`). The last variable included in our example, `page`, is stable and passive (the default behavior). Behavior information has several uses, such as to provide dummy values for writing a register when only a fraction of it (via a variable) needs to be written.

This aspect is not further described in this paper for lack of space. Some other features of Devil are not detailed here either. These features include post-actions for access paths, register and variable arrays, evaluation order for concatenation, device modes (*i.e.,* states) declarations, and conditional declarations depending on device modes.

4 Verification

Devil has been designed to express domain-specific information that is clumsy and tedious to express using the general abstraction mechanisms of GPLs. Moreover, because this information has been made explicit in the language, it enables a variety of verifications that are beyond the scope of GPLs. This section presents some of those verifications[7], both as inherent consistency constraints on a Devil specification and as use constraints of the generated device interface.

[6] These values are defined as enumerated types, not shown here.

[7] Other verifications of a specification in Devil take into account modes and conditional definitions as well as register and variables arrays. Warnings (*e.g.,* missing bit patterns in an enumerated type definition) can also be issued.

Strong typing. All entities (*e.g.,* ports, registers, variables) in Devil are strongly typed: all uses of these entities can be matched against their definition to check type correctness. This includes usage constraints for registers and variables that can only be read or written. Similar to the array indices in a GPL, various size bounds are also involved in typing: size of data that can read at given ports, size of registers, size of variables as derived from the conversion function of the associated type, size of bit masks, size of bit patterns that are given symbolic names in enumerated types, port ranges, bit ranges for register fragments.

No omission. Apart from public variables, all declared entities in Devil must be used at least once: all port arguments of a device declaration, all offsets of ranged ports, all declared registers, all bits of registers (some bits can be declared irrelevant using bit masks though), all parameters of register constructors, all private variables.

No double definition. All entities in Devil must be declared at most once: device declaration parameters, ports, registers, register constructors, parameters of register constructors, types, symbolic names and bit patterns in enumerated types, variables.

No overlapping definitions. All basic building entities of Devil (*i.e.,* ports and registers) must be used only once. More precisely, all ports must appear only once in register definitions, except when pre-actions differ. The same port can be used for reading and writing two different registers though. No bit of any register can be used to make up two different variables.

Verified use of the device interface. In addition to verifications within a Devil description, uses of the functional interface of the device by the upper layer of the driver can also give rise to other verifications: type checking (including read-only and write-only constraints) and conditional variable checking (not explained here). These verifications can be both static and dynamic.

For example, a check can be performed on a value assigned to a device variable, when it is declared as taking a restricted range of integer values. If this value is constant, the check can occur statically, otherwise code to perform this check at run time can be generated. The same verification can occur when a variable is read to ensure that the device functions properly.

Less static verifications are possible if the upper layer of the driver is written in C (as opposed to another DSL for drivers), and if it relies on a device interface generated in C. This is because the C type system is not powerful enough to express all Devil types. These verifications can be made dynamic, if a debug mode is enabled.

5 Implementation

As explained in the previous section, a specification in Devil has to be first checked for consistency. It can then be used to produce various outputs for the driver programmer: an abstract device interface provided as documentation, a library of functions (*e.g.,* in a GPL like C) to directly access the device variables, and a compiled description of the device to be used by other DSLs for programming the upper parts of the driver. In this section, we give an overview of our implementation of Devil.

The compilation process of a Devil specification is as follows. It is first transformed into an intermediate representation on which verifications are performed. The development of these verifications does not involve any novel techniques; in fact, Devil has been designed such that information exposed by the declarations make interesting properties decidable. From this intermediate representation, a compiled description of the device is generated. The compiled description is used to generate an abstract device interface (for documentation) as well as a set of C inline function definitions to access device variables. These functions possibly perform run-time type checks, if a debug mode is enabled.

The process of generating code for getting the value of a device variable is as follows: (1) from the variable definition, an ordered list of registers to read is constructed; (2) for each register, after generating its pre-actions and before generating its post-actions, an ordered list of ports to read is built; (3) invocation to bus I/O functions (or direct memory accesses) are generated to obtain a bit-string from ports; (4) using the device variable declarations, the appropriate C code (*i.e.,* bit operations) is generated to extract a specific value from the bit-string.

The algorithm for generating `set` functions is similar to the `get` case except that code must be generated to construct the bit-strings to be written in each register that the device variable is built on. However, a device variable often consists of fragments of registers, and a register is the atomic unit of information that can be read/written from/to the device. The entire value of each register needs to be constructed before performing an I/O operation. Device variable attributes (*i.e.,* trigger and passive) address this problem. That is, if a device variable has been declared *trigger* except for a specific value, this value is used to cancel the trigger effect.

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 [14]. Although successful as a proof of concept, GAL covered a restricted domain. It is this restriction which allowed us to model the domain with a unique DSL.

Recently, several computer companies (Compaq, HP, IBM, ...) have contributed to a project aimed at making device drivers source-portable across OS platforms. To do so, a Uniform Driver Interface (UDI) was defined to normalize the API between the OS and the lower part of device drivers [10]. 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 of a driver since this library already identifies the fundamental operations of these layers.

Another approach to easing driver development consists of using driver generators such as BlueWater System' WinDK [1] and NuMega's DriverWorks [2]. These two Windows95/NT-specific generators offer a graphical user interface aimed at specifying the main features of the driver to be generated. Then, a driver skeleton is produced; it consists of invocations to coarse-grained library functions. To our knowledge, no existing driver generators cover the communications with the device.

Last, we should mention languages for specifying digital circuits and systems. A standard language, widely used in this domain, is VHDL [6]. Devil differs from VHDL in that it concentrates on the communications with the device, not its inner workings.

7 Conclusion

Although devices correspond to a rapidly evolving area and require fast releases of drivers, driver development has received little (if any) attention from the research community. This situation is even more surprising when considering the level of safety that drivers should offer to guarantee the integrity of their host system.

In this paper, we have analyzed the domain of device drivers. We have pointed out that device drivers form a program family. Our study of the communication layer with devices has led us to identify key concepts of this domain. These concepts have been used to design a language, Devil, dedicated to defining the functional interface of a device.

Devil is a declarative language aimed at specifying the access mechanisms, the type and the layout of data that are exchanged to operate the device, as well as behavioral properties. Based on these declarations, thorough verifications are performed. As a result, the safety of the communications with a device is greatly improved. This approach sharply contrasts with the use of a GPL which does not permit any useful verification.

Devil has been used to specify a large number of PC devices including Ethernet, video, sound, interrupt, DMA and mouse controllers. The large spectrum of the hardware architectures of these devices has allowed us to validate the expressiveness of the language. These specifications have been used to systematically generate the implementation of the device interfaces. These interfaces have been re-introduced into the original drivers, and a preliminary assessment has not showed any significant performance penalty.

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 OS.

Availability

All Devil specifications mentioned in this paper are available from `www.irisa.fr/compose/devil`. The compiler and verifier for Devil will be made available in early January.

References

[1] BlueWater Systems, Inc. *WinDK Users Manual.* URL: www.bluewatersystems.com.

[2] Compuware NuMega. *DriverWorks User's Guide.* URL: www.numega.com.

[3] 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* 10^{th} *International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in Lecture Notes in Computer Science, pages 170–194, Pisa, Italy, September 1998.

[4] D. Cuka and D. Weiss. Engineering domains: Executable commands as an example. In *Proc. Fifth Internationbal Conference on Software Reuse*, June 1998.

[5] Edwaed N. Dekker and Joseph M. Newcomer. *Developing Windows NT device drivers : A programmer's handbook.* Addison-Wesley, first edition, March 1999.

[6] IEEE Standards. *1076-1993 Standard VHDL Language Reference Manual*, 1994. URL: standards.ieee.org.

[7] National Semiconductor. *DP83905: AT/LANTIC AT Local Area Network Twisted-Pair Interface.* URL: www.national.com.

[8] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9, mar 1976.

[9] Rubén Prieto-Díaz. Domain analysis: An introduction. *Software Engineering Notes*, 15(2), April 1990.

[10] Project UDI. *UDI Specifications, Version 1.0*, September 1999. URL: www.project-udi.org.

[11] Alessandro Rubini. *Linux Device Drivers.* O'Reilly, first edition, February 1998.

[12] S. Thibault and C. Consel. A framework of application generator design. In M. Harandi, editor, *Proceedings of the Symposium on Software Reusability*, pages 131–135, Boston, Massachusetts, USA, May 1997. Software Engineering Notes, 22(3).

[13] 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.

[14] 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.

A Measuring the Scope of Devil

To assess how much of a device driver is covered by Devil, the code devoted to device communications needs to be measured in existing drivers. However, this code typically consists of expressions combining many bit operators. As a result, the source code is not an accurate basis for this assessment. To circumvent this problem, we performed our measurements directly on the assembly code, where bit expressions are decomposed.

Our approach to performing these measurements is as follows. We took some existing Linux drivers, labeled the source fragments devoted to device communication, and compiled the code for Pentium. Then, we counted the labeled instructions in the resulting code, and computed a ratio for each driver (see table below).

Device Domain	Linux Driver	Measured Ratio
Ethernet Controller	3c59x.c	20.6 %
Ethernet Controller	8390.c	24.2 %
Mouse Controller	busmouse.c	16.2 %
Mouse Controller	msbusmouse.c	21.0 %
SCSI Controller	am53c974.c	25.1 %
SCSI Controller	ncr53c9x.c	21.6 %
Sound Chip	ad1848.c	30.7 %
Sound Chip	cs4232.c	21.1 %
Sound Chip	es1371.c	19.0 %

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399