The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

A Domain Specific Language for Video Device Drivers: from Design to Implementation

Scott Thibault, Renaud Marlet, and Charles Consel
IRISA / INRIA - Université de Rennes

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

A Domain Specific Language for Video Device Drivers: from Design to Implementation

Scott Thibault Renaud Marlet Charles Consel

IRISA / INRIA - Université de Rennes 1
Campus Universitaire de Beaulieu
35042 Rennes cedex, France
{sthibaul,marlet,consel}@irisa.fr
http://www.irisa.fr/compose *

Abstract

Domain-specific languages (DSL) have many potential advantages in terms of software engineering ranging from increased productivity to the application of formal methods. Although they have been used in practice for decades, there has been little study of methodology or implementation tools for the DSL approach. In this paper we present our DSL approach and its application to a realistic application: video display device drivers.

The presentation focuses on the validation of our proposed framework for domain-specific languages, which provides automatic generation of efficient implementations of DSL programs. Additionally, we describe an example of a complete DSL for video display adaptors and the benefits of the DSL approach in this application. This demonstrates some of the generally claimed benefits of using DSLs: increased productivity, higher-level abstraction, and easier verification. The DSL has been fully implemented with our approach and is available [1].

1 Introduction

In contrast to a general purpose language (GPL), a *domain-specific language* (DSL) is a language that is expressive uniquely over the specific features of programs in a given problem domain. It is often small and declarative; it may be textual or graphic. DSLs have also been called *application domain languages* [6], *little* or *micro-languages* [2], and are related to scripting languages [18]. DSLs have been used in various domains such as financial products [1], telephone switching systems [11, 16], operating systems [19], and robot languages [5]. Languages such as SQL, TEX and shells may also be considered DSLs.

Software architectures based on DSLs primarily aim at achieving faster development of safer applications. Because constructs in a DSL abstract key concepts of the domain, the developer (that does not have to be a skilled programmer) can write more concise and higher level programs in less time. Programming with a DSL also contributes to safety because it is less error-prone than with a GPL. Additionally, high-level constructs translate, in practice, into the reuse of validated components. Moreover, when the language is small and specific, it is possible and easier to build automatic validation and test generation tools. For example, termination properties may be considered if the language is not Turing-complete.

A DSL may also be seen as a way to parameterize a generic application or to designate a member of a program family. A program family is a set of programs that share enough characteristics that it is worthwhile to study them as a whole. In fact, designing a DSL actually involves the same *commonality analysis* [11] that is used in the study of a program family: assumptions that are true for all members of the family and variations among members. This process should be performed both by domain experts and software engineers.

Though actual uses of DSLs record benefits such as productivity, reliability and flexibility [15], implementing DSLs is often difficult and costly [7]. There are two main approaches to language implementation, each with significant disadvantages: those that are based on compilers (translation from

*This work has been partly supported by FRANCE TELECOM contract CNET 96-1B-027 and DARPA contract F19628-95-C-0193.

[1] http://www.irisa.fr/compose/dsl/genprog.html

the DSL to a target machine or GPL) are not easy to write or to extend, and extensions require skills in compiler technology that cannot be expected from "domain developers"; those that are based on interpreters are easier to write or to extend but are less efficient [4]. This issue also impacts maintainability [21] because complexity in the compiler defeats the software engineering goals of using a DSL. Depending on objectives, either one or the other style of implementation is thus chosen: application generator or interpreter.

We have proposed a framework for the development of application generators that reconciles both alternatives [20]. It relies on *partial evaluation* [12, 14], a program transformation technique that is well suited to automatically transform interpreters into compilers [13]. Partial evaluation exploits known information about a program's input to be able to evaluate parts of a program in advance. Given a program and the known portion of its input, a partial evaluator produces a specialized program. In this new semantically equivalent program, computations depending on known values have already been performed.

Our framework is structured into two levels. The first level consists of the definition of an abstract machine, whose operations can be viewed as generic components that capture important operations of the domain. The second level is the definition of a micro-language in terms of the abstract machine operations, thus providing a high level interface to the abstract machine. The use of partial evaluation in our framework is twofold, corresponding to each level: it maps an abstract machine into an efficient implementation, and a micro-program into an abstract machine program. The development of this framework is supported by industry partners for realistic applications.

This paper describes a realistic application of our framework for the automatic generation of video card drivers. This domain naturally forms a program family, for which DSLs are well suited. We present the design and definition of a complete DSL for video display adaptors. Concerning performance, we show how partial evaluation can yield efficient drivers. Concerning safety, we insure that all generated drivers can be proven to terminate and define some analyses that can greatly improve their reliability.

Our contributions can be summarized as follows:

- We validate our framework of application generator design on a realistic example: video card device drivers.

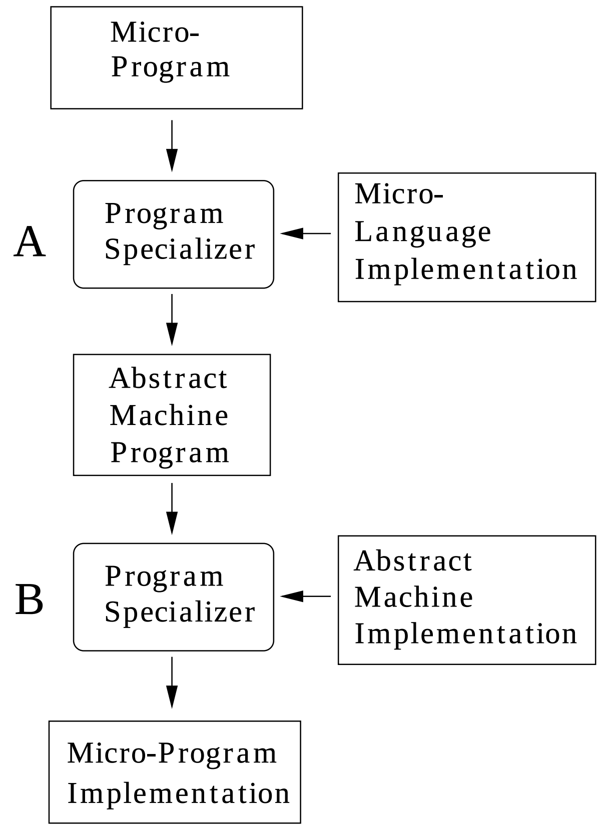

Figure 1: Generic program instantiation.

- We define a DSL for generating such drivers. This restricted language allows program verifications.

- We provide a flexible implementation of this language that generates efficient video drivers.

- We illustrate the benefits of DSLs as a software architecture.

The rest of the paper is organized as follows. Section 2 describes our framework for application generator design in further detail. Section 3 presents the domain of video card drivers. Section 4 describes the two-level design: abstract machine and graphics adaptor language. Section 5 discusses the results of applying this approach to the domain of video drivers. Section 6 summarizes the results of this experiment and identifies future work, both for the language and the framework.

2 The DSL Framework

In a previous paper [20] we presented an approach to generic software design. In this approach, we consider the implementation of a program family as a generic program. The parameterization of this generic program corresponds to variations within the program family and can be represented using a micro-language. A *micro-language* is a small restricted DSL which formally defines the program family (an instance of the program family is specified by a micro-program) and is restricted to allow analysis. The generic program implementing a program family is constructed in two layers, and automatic instantiation is performed by program specialization (i.e., partial evaluation), as illustrated in Figure 1. Together, the two applications of program specialization provide a complete path from a micro-program to an efficient implementation.

The abstract machine is the definition of the fundamental *operations* of the domain that are used to implement members of the program family. The abstract machine is implemented as a highly parameterized library, whose parameters represent *operational* variations within the domain. Any given abstract machine program provides values for operation parameters that indicate the desired functionality. A partial evaluator can eliminate the genericity from the library functions using these known values to produce an efficient implementation, as shown in the bottom section of Figure 1.

The micro-language captures the variations within the program family in terms of *what* family members do, as opposed to *how* family members operate, which is captured by the abstract machine. The micro-language is implemented by an interpreter, which invokes abstract machine operations by calling the corresponding library subprograms. The micro-language provides an interface to the abstract machine, and the interpreter implements a mapping from a micro-program to the abstract machine. This mapping depends only on the micro-program such that, given a micro-program, a partial evaluator, with the micro-program as known input, eliminates all operations in the interpreter leaving only the remaining calls to the abstract machine. Thus, a partial evaluator can produce an abstract machine program from a given micro-program, as shown in the top section of Figure 1.

For the application of this approach we use a partial evaluation system named Tempo [8]. Tempo is a fully automatic partial evaluator for C. Users of Tempo specify inputs to the entry point and global variables as either known or unknown. In our approach, we insure the successful application of partial evaluation via the separation of the abstract machine and the interpreter, each having its on state represented in C by global variables. The interpreter state is specified as known and the abstract machine state is specified as unknown. The following simple rules guarantee correct separation and thus successful automatic partial evaluation.

1. Interpreter subprograms may only use variables of the abstract machine state as actual parameters of a subprogram call.
2. Abstract machine subprograms may not contain any references to the interpreter state.

3 Video Driver Domain

This section introduces the domain of the experiment: video adaptor device drivers. A *video adaptor* (or *video card*) is a hardware component of a computer system which stores and produces images on the display. Video cards consist of a frame buffer, and a graphics controller. The *frame buffer* is a bank of high speed memory used to store the display data, including the currently visible image. The *graphics controller* consists of two main functionalities: producing the video signal for the display, and providing access to the frame buffer to create the display image. Graphics controllers all provide similar sets of functionalities (e.g., changing the display resolution).

Although all adaptors provide similar functionalities, their programming interface is different from vendor to vendor, and often between successive models of the same adaptor. This is true of most devices, and is resolved by the use of device drivers. Device drivers generally consist of a library of functions that implement a standard API that is fixed for all devices. Thus the driver's purpose is to translate the standard API operations into the operations required by a specific device, providing a uniform interface to the operating system and applications.

Video device drivers provide two main services to the operating system and applications. The first is to put the graphics display into different video modes. A *video mode* (or *graphics mode*) is defined by the horizontal and vertical resolution, the number of colors per pixel and screen refresh rates. The second service provided by the driver is to provide access to hardware drawing operations. For example, most video cards provide line drawing hardware, which draws lines on the display at a much faster rate than would be possible in software.

4 Application of the Approach

We have applied the approach described in section 2 to a family of device drivers for video adaptors. We considered an already existing set of device drivers from the XFree86 X Window server created by The XFree86 Project, Inc. [23]. The XFree86 SVGA server is a generic X Window server, written in C, supporting several different cards using a device driver architecture. This server contains drivers for cards from about 25 different vendors. Additionally, each driver supports as many as 24 different models from the same company. This structure alone indicates that there is enough similarities between models of the same vendor to implement them as a generic program, but that it is not reasonable to do so for multiple vendors. This may be due to efficiency, but more likely is due to the lack of a methodology to handle larger scales of variation.

The remainder of this section details the application of our approach to the construction of an application generator of video drivers (for different vendors) for the X Window server. We first discuss the definition of an abstract machine for the domain, identified by studying the existing device drivers. Then we describe a DSL for generating video drivers and related design issues.

4.1 The Abstract Machine

The abstract machine for the video driver domain was designed primarily by studying the implementation of existing drivers. The abstract machine was also iteratively refined during the development of a DSL. We identified three patterns which appeared in the existing drivers that could be used to guide the definition of abstract machine operations.

Operation pattern. The first of these patterns corresponds to simple atomic operations in the abstract machine. There are two forms in which this pattern appears: as repeated fragments of code that differ only by data, and as fragments which perform the same treatment but have a small number of variations on how it is performed. In the first case, the fragments are often already identified and placed in a library or defined as a macro. These fragments directly correspond to abstract machine operations.

As an example of the second case, the device drivers are dominated by occurrences of code fragments which read or write data from or to the video card. Communication with hardware devices can be handled in a small number of different ways, and the scheme chosen varies from vendor to vendor. There were several occurrences of three of these different schemes of I/O, differing only in certain data (e.g., the I/O address). These fragments were captured in a single abstract machine operation defined as follows:

```
write_port(port_number: integer,
           index: integer,
           indexed: boolean,
           pair: boolean,
           pci: boolean)
```

This instruction is parameterized by flags to specify which scheme to use (indexed, paired, or PCI), and the data used by the scheme to perform the I/O (port_number, index).

Combination of operations pattern. The second type of pattern recognized can be identified as expressions or combinations of operations. This pattern is characterized by expressions or combinations of operations that have no commonalities between members of the family. For example, in the device drivers there are sequences of shifts and logical expressions which are different for every driver. Although there are no commonalities in those expressions from one driver to the next, we can identify a sufficient set of operations to construct any instance. The selection of these operations depends not only on the existing samples, but on an understanding of the domain, and speculation on the future of the domain.

The following code fragment shows an example of this pattern from one of the existing drivers.

```
outb(0x3C2, ( inb(0x3CC) & 0xF3) |
            ((no << 2) & 0x0C));
outb(OTI_INDEX, OTI_MISC);
outw(OTI_INDEX, OTI_MISC |
     ((( inb(OTI_R_W) & 0xDF ) |
      (( no & 4) << 3)) << 8));
```

This portion of the driver maps the value of `no` onto the appropriate registers in order to select the clock. For a given driver, there may be any number of reads, writes, shifts and logic operations, but no other operations. Thus, we can implement any given driver with a sequential composition of a small number of abstract machine operations.

Control pattern. The last pattern consists of code fragments that share a common control structure, but contain code fragments matching the combination of operations pattern previously discussed. For example, consider a function of the device driver which is used to save, restore, and set the clock value on the video card.[2] This function has the following form:

```
Bool ClockSelect(int no)
{
  switch (no) {
  /* Save the clock value. */
  case CLK_REG_SAVE:
    Series of I/Os and logic operations.
    break;
  /* Restore the saved clock value. */
  case CLK_REG_RESTORE:
    A second series of I/Os and logic operations.
    break;
  /* Set the clock value to no. */
  default:
    A third series of I/Os and logic operations.
  }
}
```

The series of I/Os and logic operations in this example follow the combination of operations pattern, and can be constructed by sequences of abstract machine operations.

For this pattern, we introduce higher-order abstract machine operations. That is, abstract machine operations which take sequences of abstract machine operations as parameters. These parameters correspond to the contained fragments that follow the combination of operations pattern. The example above is captured by the following abstract machine operation:

```
change_clock(save_clk: instructions,
             restore_clk: instructions,
             set_clk: instructions)
```

Conclusion. Using these patterns with existing examples, we were able to define an abstract machine that could express the behavior of any particular device driver. Although they were typically easy to recognize, it is important to realize that it was necessary to abstract from certain details in order to see the different patterns. E.g., in our experiment, the examples were mostly written by different people, who had different styles of programming, and sometimes took different approaches to the same problem. In this situation, it was necessary to determine if the same functionality could be implemented with a common structure, which happened to always be the case.

[2]Video cards have programmable clocks which can be set to different frequencies to control the video refresh rate.

4.2 The GAL Language

In this section we present the Graphics Adaptor Language (GAL) for video device driver specification. In order to understand where the language comes from, it is important to know what the essential variations are among video adaptors. The remainder of the section describes the variations that exist between cards and the corresponding constructs in GAL that capture them. A complete example of a GAL specification is described in Appendix A.

4.2.1 Ports, Registers, Fields and Params

A video adaptor is controlled by setting certain parameters stored in hardware registers of the card. These registers have addresses. A single parameter may be stored in multiple registers and only certain bits of the registers may be used. Thus the layout of the parameters on the register space is the first major variation between cards.

Access to the registers are provided through various communication schemes. As mentioned in the previous section, there is a small number of different schemes that can be used to communicate with a hardware device from a program. The choice of communication scheme is the second major variation between cards. We define several concepts to describe these notions of communication and register layout.

Ports. The first concept is the *port* which is used to define a point of communication. For example, the declaration

```
port svga indexed:=0x3d4;
```

defines a port named `svga`, which uses an indexed communication scheme at the I/O address 0x3d4. This is a standard port used by many video cards.

Registers. A second concept is provided by the *register* declaration, which defines how to access

Standard field	**Purpose**
HTotal, HEndDisplay, HStartBlank, HEndBlank	Horizontal resolution settings.
VTotal, VEndDisplay, VStartBlank, VEndBlank	Vertical resolution settings.
LogicalWidth	Width of virtual screen.
StartAddress	Display start address.
ClockSelect	Clock selection.
Standard param	**Purpose**
RamSize	Frame buffer memory size.
LinearBase	Address of linear space.
LinearAperture	Size of linear space.
NoClocks	Number of fixed clocks.

Table 1: Predefined fields and params.

registers on the card using the defined ports. For example, the declaration

```
register ChipID:=svga(0x30);
```

defines a register `ChipID`, which is accessed through port `svga`, at index 0x30.

Fields. The next concept is specified with a field declaration. The *field* declaration defines where a logical value is stored (in which bits of what registers) and a mapping from logical values to actual stored values. For example, the declaration

```
field LogicalWidth:=
  Control2[5..4] # Offset scaled 8;
```

defines a field `LogicalWidth`, which is stored in bits 5 and 4 of the `Control2` register and the entire `Offset` register. Additionally, the mapping clause (`scaled 8`) specifies that the value stored in the register is $\frac{1}{8}^{th}$ the actual value. The mapping is needed because cards often store a value which is some function of the field's actual value.

Parameters. Related to the field declaration, the *parameter* declaration is the definition of a constant value that is either explicit in the specification or read from the card during configuration. An example of the former case would be

```
param NoClocks:=4;
```

The majority of a GAL specification consists of the definition of fields for standard values that are used to control the video adaptors and parameters which determine certain features of the card (e.g., size of the frame buffer). Table 1 lists some of these predefined field and parameter names that can be defined in GAL specifications.

4.2.2 Clocks

A third major variation between different adaptors is the use of clocks. All adaptors have a clock which controls the frequency at which data is sent to the display. This frequency needs to be changed for different resolutions, and there are two approaches to doing this. One is to have a fixed number of frequencies to choose from, and the other is to have a programmable chip that can generate many frequencies by changing its parameters. The cards with a fixed number of clocks vary in the number of clocks and the frequencies provided, while the cards with a programmable clock vary in how the clock is programmed and its range of frequencies.

A card that has fixed clocks can be specified by defining a parameter `NoClocks` and a field `ClockSelect`. The `NoClocks` constant defines the number of clocks available, and the `ClockSelect` field defines the field which selects the clock.

For cards that have programmable clocks, a special construct is defined to specify how to program the clock. For example,

```
clock f3 is 14318*f3M / (f3N1*f3N2);
```

defines a clock named `f3`, which is programmable according to the equation on the right. The equation defines the frequency generated based on programmable values, which are defined elsewhere by the three fields `f3M`, `f3N1`, and `f3N2`. Given the desired clock frequency, the device driver uses the specified equation to find values of `f3M`, `f3N1`, and `f3N2` which approximate this frequency as closely as possible.

4.2.3 Identification

The fourth major variation observed among video cards is how the card is identified. This information is required for systems which dynamically configure themselves to use whatever card is available at that time. Card identification uses a small number of predicates which test the card and follows a decision tree to decide if the card is supported by the driver and which one.[3] Thus, we define an appropriate construct for specifying this type of decision tree in GAL.

The following is an example of this identification construct.

```
identification begin
 1: writable(Segment) => (true=>step 2);
 2: Chip_id=>(1=>oti087,others=>step 3);
 3: Chip_id2=>(0=>oti037c, 2=>oti067,
               5=>oti077);
end identification;
```

This example identifies one of four models (oti037c, oti067, oti077, oti087) of cards that use an OTI graphics controller. The construct defines a series of steps numbered 1-3 to the left. At each step, the expression to the left of the arrow is evaluated and the result is compared to the list of decisions on the right. If no decision is matched on the right, then identification fails and indicates that the driver does not support the card. Possible decisions are to identify the card or proceed to another step. Step 2, for example, reads the value of the `Chip_id` register, and if the result is 1, identifies that an oti087 is present, otherwise proceeds to step 3 for further tests. The stepwise syntax reflects the way diagnostic procedures are commonly described in manuals.

4.2.4 Modes

The final major variation between cards is that many adaptors require some flags be set under certain operating conditions. These are referred to as *modes of operation* in GAL, and are defined with the mode construct. The *mode* construct is used to specify a predicate and a sequence of assignments to fields, which enable or disable the corresponding mode of operation for the video card. For example,

```
mode HighRes:=HTotal>800;
enable HighRes sequence is Control[5]<=1;
```

This mode declaration defines a mode, `HighRes`, which indicates that a '1' must be stored in bit 5 of `Control` in order to use a video mode in which the horizontal resolution is greater than 800 pixels. In our implementation, the predicate `HTotal>800` is tested after changing the video mode; if it is true, the sequence `Control[5]<=1` is executed.

In addition to user defined modes, there are also a few built-in modes. The built-in modes have fixed predicates, but allow the specification of enabling and disabling sequences. For example, the built-in mode `SVGAMode` is true for all graphics modes and thus the user-defined enabling sequence is executed each time the mode is changed.

[3] One device driver often supports multiple cards from the same vendor.

4.2.5 Run-time variations

In addition to the variations that exist between cards, there are variations within a single driver that depend on conditions not known until run-time (of the driver). For example, some video adaptors operate differently depending on the hardware bus utilized (AT, PCI, or VLB). Additionally, if one wants to build a single device driver for a number of models from the same vendor, the variation between those models has to be chosen at run-time. In GAL, the `cases` construct is used to describe this type of variation.

As an example, the following statement is used to define the clocks for different models of S3 cards.

```
cases
for S3_TRIO32,S3_TRIO64
  field ClockSelect:=Miscr[3..2];
for others
  field ClockSelect:=Control[3..0];
end;
```

This example specifies that if the card identified at run-time is a S3_TRIO32 or S3_TRIO64, then the card has four fixed clocks selected by bits 3 and 2 of the `Miscr` field. All other cards have sixteen clocks selected by bits 3 down to 0 of the `Control` field.

4.3 Design of GAL

This section discuses some of the many forces that influenced the design of GAL. The first two subsections describe two main inputs to the design process: a definition of variations in the family and knowledge about the domain. In our case, the domain knowledge came from existing documentation used by domain engineers. Other important issues are the level of abstraction, the level of restriction, readability, maintainability, etc. While the level of abstraction and the level of restriction are of particular importance for DSLs, issues like readability and maintainability apply to both DSLs and GPLs

4.3.1 Defining Variations

One of the main inputs to the design of a DSL is a description of the variations that exist among the target set of applications. The defined variations imply requirements on the DSL in order to distinguish among instances of the program family. In our case, these variations came from a study of the documentation of existing video cards. In addition to studying different cards, inspection of the existing device drivers provided a more detailed source of variations at the implementation level. For example, given that there were a small number of ways to communicate, which varied among cards, there must be some construct in GAL specifications, which would allow the selection of the correct communication scheme. Some of this information can also be extracted from the parameters of the abstract machine operations.

4.3.2 Domain knowledge

The other main input to the DSL design process is knowledge of the domain in terms of the abstract objects or concepts and terminology used in the domain. This knowledge may come from a domain expert or from existing natural language specifications, as in our experiment. This is an important input because it leads to a more abstract user-level DSL. An appropriate terminology provides a DSL that is familiar to people of the domain. The identified abstract objects that are affected by variations in the program family provide starting points for declarative constructs.

In this experiment, we looked at several English specifications of video cards to identify the concepts and terminology used within the domain. The clocks, ports and registers are examples of concepts in the domain that we identified. After identifying them, we considered what attributes of the objects were related to variations within the program family. Declarative statements were then defined to specify the values for the attributes that varied. Thus, the abstract objects identified in our experiment directly translated to declarative constructs in the DSL. Additionally, the relationship between the objects translated into a reference relationship in the DSL. For example, registers are defined by references to port definitions. This may suggest the use of an object-oriented analysis for DSL design.

4.3.3 Level of Abstraction

One of the most important goals guiding the DSL is to provide a high-level of abstraction. In particular, we wish to intentionally focus on raising the level of abstraction from the abstract machine level. In fact, it may be desirable to include information in the DSL, which is not even used for implementation, but may be used in analyses or for documentation.

As an example of abstraction, the abstract machine developed for the video device drivers includes operations for doing bitwise shifts and logical operations. However, these types of expressions do not appear in GAL because we intentionally introduced the idea of fields and parameters to eliminate the low-level procedural nature of these expressions. This also eliminates a common source of errors.

After a preliminary design of the language, the language and abstract machine are revised in an iterative way. The revision process must satisfy the correspondence constraint between the language and abstract machine: it must be feasible to provide a mapping from the language to the operations of the abstract machine as an interpreter. During this revision process the level of abstraction must also be considered. Although it is possible to move all of the functionality of the language into the abstract machine, making the mapping essentially one-to-one, there must be conscious decisions made about where to draw the line between the interpreter and the abstract machine. The primary consideration here is the separation of functionality from specification. The abstract machine should specify how applications in the family are implemented. The interpreter, on the other hand, should specify how to make the design decisions required to map a design specification (i.e., DSL program) into an implementation (i.e., abstract machine operators).

4.3.4 Level of Restriction

Another major concern is restricting the language. It is important to consider what types of analyses might be performed on specifications in the DSL in order to insure that the language is restricted enough to make the analyses feasible. For example, in the GAL language we have intentionally not introduced loops, which insures that all device drivers can be proven to terminate. Additionally, we perform other analyses to detect common errors in the specification by providing explicit information that is difficult or impossible to extract from general purpose languages. An example of this is checking that the bits of each register belong at most to one field. This information could not be retrieved, in general, from a driver implemented in a language such as C.

4.3.5 GPL principles

In addition to the design goals that are specific to DSLs, there are several principles of general purpose language design that also apply to DSL design. General purpose languages can also help DSL design by providing a standard set of constructs that may be restricted for use in the DSL, but would still be recognized as a common construct.

On the other hand, the `cases` construct introduced in GAL is an interesting example of a construct which possibly has applications in DSLs in general (when a predefined abstraction may, conditionally, have one of several definitions), but is not useful for GPLs, since the behavior is totally described by the program itself and abstractions are explicitly invoked. One of the main purposes of introducing a DSL and an application generator is to embed knowledge about how to implement certain operations of the domain into the application generator. As a result, there are often declarative constructs in DSLs that are translated into executable code by the application generator, which is not generally true of general purpose languages. Since these declarations really imply operations, there is often a need to make choices between the implied operations that can only be made at run-time. This leads to the type of dynamic selection of multiple definitions that is provided by the `cases` statement. Since a main motivation of utilizing a DSL is to raise the level of abstraction, it will be common for DSLs to have declarative objects which imply operations and require this dynamic selection. Thus, we suspect that this construct will be useful in DSLs in general, and in fact have found it necessary in other DSLs that we have experimented with. This suggests that there are new constructs and principles that are interesting and unique to DSLs and warrant study.

5 Results

In this section we present the results of applying our framework to the domain of video device drivers. The results are presented in terms of the advantages we have gained from using our approach for this family of drivers. There are two aspects of the approach that led to these advantages. One aspect is the use of DSLs and application generators in general, and the second is specific to our framework for application generator design.

5.1 Domain Specific Language

The GAL language demonstrates many advantages of using an application generator with a DSL for the video device driver domain. These benefits include an increased level of abstraction, the possibility of automated program analyses, reuse, and productivity.

```
Profile of S3_TRIO64 + S3_TRIO32
Maximum resolution: 4088x2047
Maximum virtual screen: 3328x2520
      (with maximum RAM)
Ram size: 512k-8192k
Clock range: 135-270MHz
Resolution limited to 2304x1728 by
      the clock (max. refresh 67Hz).
```

Figure 2: An extract of generated S3 card profile.

There are two significant examples of the benefit of a higher level of abstraction. The first, already discussed in section 4.3.3, is the use of ports, registers, and fields to abstract from the low-level bitwise operations that would otherwise have to be used. This eliminates many common errors, is more readable, and easier to write. A second example is an abstraction from implementation. The X Window server can be considered a framework, where the device driver provides the additional functions. As with any framework, the device driver needs to be implemented in a certain way in order to be compatible with the server and requires considerable knowledge about the framework. Using an application generator, knowledge about the framework and compatibility issues are coded in the application generator, and hidden from the designer.

GAL also demonstrates that automatic analyses can be performed on the DSL, which would not be possible or feasible with a general purpose language. Example analyses that are performed on GAL specifications include detecting unused definitions, checking for exhaustive identification of video cards, identifying overlap in field definitions, checking for minimum requirements on predefined fields, and generating a card profile (summary of card characteristics). None of these analyses would have been feasible on the existing device drivers implemented in C. Using GAL not only makes the analyses feasible, but also easy to implement. For example, all of these analyses for GAL were implemented within a single day.

One particularly interesting analysis is the one which generates a card profile. Generating a card profile is an analysis which, from the GAL specification, produces a summary of the video modes that are supported by the generated device driver.

Figure 2 shows an extract of the profile generated for the S3 specification listed in Appendix B. A profile is generated for each subset of cards in the specification that have the same profile. The figure shows the profile for the `S3_TRIO64` and `S3_TRIO32`. This summary can be compared with vendor specifications to find mistakes in field definitions and provides automatic documentation of the specification.

Finally, using an application generator provides reuse by capturing design knowledge. In the domain of video device drivers there are large benefits of reuse because there is a large growing number of video cards which could potentially be generated from a single application generator. The amount of productivity gained depends on the ease of building the application generator and consequently on the approach to its design. Thus, we discuss productivity measurements in the next section with respect to our framework.

5.2 Our Framework

In addition to the advantages obtained from the DSL approach, there are several advantages demonstrated by GAL due to our framework of generator design. The experiment shows that the framework achieves automatic and predictable generation of efficient video drivers, and a high-level of reuse. GAL also demonstrates that the benefits of the two-level approach for analyses and multiple implementations are of practical value.

5.2.1 Reuse and Productivity

The abstract machine for X Window device drivers consists of 95 small C procedures totaling 1200 lines. Implementing the abstract machine has roughly the same difficulty level as implementing a single driver directly, as the code is very similar. Since we had existing device driver implementations, some of the abstract machine code could be reused from those drivers.

The interpreter for GAL consists of 4300 lines of C code and an automatically generated parser, much of which concerns building an environment and look-up routines for declarations. Thus, together the system consists of about 5500 lines of C code. We can compare this to the size of the existing hand-coded drivers which averaged about 1500 lines. Though the effort required to build an interpreter should be less than that for building a device driver, we can estimate that the application generator requires a little more than 3.5 times the effort of an individual driver (assuming code size proportional to effort).

For the version of the X Window server we used, the existing drivers together consisted of 35000 lines of code. The GAL specifications that have been written are at least a factor of 9 smaller than the corresponding existing C driver . We can then estimate that these drivers could be generated from less than 4000 lines of GAL specifications plus the 5500 lines of the generator, totaling less than 10000 lines. This is an estimated productivity gain of a factor of 3.5. In practice there would be a higher gain, since GAL specifications are easier to write then the corresponding C driver. In addition, having an interpreter for GAL provides a prototyping environment.

5.2.2 Efficiency

Here we consider two measures of efficiency: object code size and execution speed. Although designing an interpreter is easier than designing a compiler, there are significant losses in speed and size (compared to compilation). In terms of speed, interpreters are typically 10-100 times slower than compiled programs, and in terms of size, our GAL interpreter is 10 times larger than a typical driver in object code size. However, a benefit of using partial evaluation is that we can regain the loss in efficiency.

We used Tempo [8], a partial evaluator for C, as the program specializer used to translate GAL specifications to abstract machine programs, and to produce an efficient implementation of the abstract machine programs. In order to make a size comparison, we compared the object file sizes of the generated drivers to that of the hand-coded drivers. On average, the generated driver is only 30% larger than the hand-coded one.

The speed of most of the device driver functions are insignificant, as they are only called during configuration. However, we picked three device driver functions used for drawing lines and rectangles in hardware to benchmark performance. Since the interpreter level of our framework is guaranteed to be eliminated (see section 2), we are only concerned with the abstract machine layer.

For comparison, we prepared three versions of the X Window server for an S3 TRIO64V+ video card on a Pentium PRO-200. Table 2 shows the timing results for the three servers. The S3 XAA server is the X Window server provided with XFree86 and the included hand-coded S3 device driver. S3 AM

Server	lines/s	percent
S3 XAA	189,000	100
S3 AM	150,000	79
S3 PE	191,000	100
Server	rectangles/s	percent
S3 XAA	203,000	100
S3 AM	169,000	83
S3 PE	205,000	101

Table 2: Performance results.

is the same server with a device driver which directly uses the abstract machine. Finally, S3 PE is the same server using the abstract machine, but after partial evaluation. The table shows the performance of these servers for lines and filled rectangles of size 10 as measured by the standard XBench benchmark utility.[4] The table also includes a percentage using S3 XAA as a baseline.

The table indicates that there is a loss of about 20% in performance from the use of the abstract machine. This loss of performance can be contributed to error checking, interpretation, function call, and data copying overhead. Data copying is due to the need to communicate across abstract machine operations. The write operation includes error checking to insure that if previous operations fail the resulting data is not written to the card. This is particularly important because the card could otherwise be damaged. Finally, the I/O operations require some interpretation of their parameters to determine the type of I/O to perform and which addresses to use. Although directly using the abstract machine incurs this performance loss, the results for the S3 PE server show that the program transformations performed by partial evaluation are able to recapture all of the performance loss. A majority of the error checking can also be eliminated using Tempo because often the operations preceding write operations can not fail, and thus error conditions do not need to be checked. Finally, the parameters which are interpreted to select the type of I/O to perform and used for address computation are known and eliminated by Tempo. Tempo also performs inlining and copy elimination which eliminates function call and data copying overhead.

[4] A small size is used to minimize the effect of the hardware.

5.2.3 Analyses

Our framework for application generator design contributes in two ways to the use of program analyses. The generation process is predictable and can be analyzed, and the separation of the abstract machine from the interpreter allows analysis at the abstract machine level.

As an example, the GAL abstract machine includes operations that allocate and deallocate temporary storage and operations which use the temporary storage. As long as the operations which use the temporary storage are only used between a set of allocate and deallocate operations, we can insure there will be no uninitialized pointer dereferences. The analyses of partial evaluation are capable of producing a specification of all the programs that could possibly be generated by the partial evaluation process. From this, we can obtain a formal description of all possible abstract machine programs that could be generated, and can check that the operations are always generated in the correct order. Thus, for the GAL system we can prove that uninitialized pointer dereferences will never occur. This description of the generation process may also be analyzed for performance properties, for example.

The separation of the abstract machine and the DSL provides an intermediate level at which analyses can be performed and could allow analysis at run-time. In fact, this separation corresponds to a standard technique of program specification, which factors the verification process into two parts [3]. As an example of analysis at run-time, we may wish to check that device access within a video driver is safe (e.g., does not access the disk device). This cannot be done until run-time because it depends on what devices are present at run-time. In this case, we might accept video drivers in abstract machine form and analyze the abstract machine at run-time. Partial evaluation can be performed at run-time [9], so the efficiency can still be recaptured. This kind of analysis is not feasible on machine code or even Java bytecodes due to their general purpose nature. In proof-carrying code [17], the burden of proof is put on the programmer and the proof is sent with the code to be verified (verification being easier), whereas here we make the proof easier so that it can be done at run-time.

5.2.4 Multiple implementations

The video device driver family also demonstrates a useful application of having multiple implementations of interpreters and abstract machines. In this

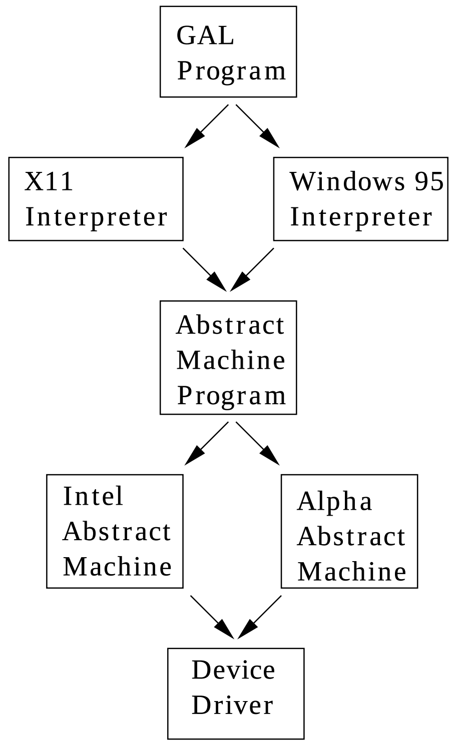

Figure 3: Multiple implementations.

domain, it would be desirable to have abstract machines for several architectures and interpreters for different operating systems. For example, Figure 3 shows the situation where there are implementations of interpreters for Microsoft Windows 95 and Linux/X11, and implementations of the abstract machine for the Dec Alpha and Intel based computers. In this situation, with the equivalent of two application generators (interpreter/abstract machine pairs), the same GAL specification can be used to generate four different device drivers. We have implemented the X11/Intel path of Figure 3.

For prototyping, we have also benefited from having a second implementation of the abstract machine which simulates the abstract machine operations. The simulation records the values that would be written to the card by the real abstract machine. This is an important feature as some video adaptors can be damaged by writing inappropriate values to the card.

6 Conclusions and Future Work

Domain specific languages hold the promise of delivering high payoffs in terms of software reuse, automatic program analysis, and software engineering. In this paper we have presented GAL, an example of a complete DSL for a realistic program family: video device drivers. We also demonstrated the benefits of DSLs by showing how GAL raises the level of abstraction of device driver specifications and identifying some analyses that can be performed on GAL specifications because it is domain specific.

A further contribution of the paper is to validate our framework of application generator design by applying it to this program family to provide an implementation of GAL. Since our implementation is based on partial evaluation, it provides a complete interpreter for prototyping device drivers, but still automatically generates efficient device drivers. Efficiency is demonstrated with results comparing hand-coded drivers to automatically generated device drivers. Generated drivers are roughly one third larger than hand-code drivers and perform equivalently in terms of speed. Additionally, we give measures on expected reuse benefits; GAL specifications are roughly a factor of 9 smaller than a driver hand-coded in C.

Although our framework significantly reduces the development time of application generators, future work could be done in this direction. Specifically, this approach would benefit from a generator-specific reuse method that would allow interpreters and abstract machines to be constructed from reused composable parts. Additionally, given the nature of DSLs, they are extended frequently to adapt to new program requirements, and the ease of extension also needs to be considered for such language components.

Our implementation of the static analyses indicates that methods of quickly constructing static analyses should also be investigated (e.g., composable analyses). This is more important for DSLs than GPLs, since static analyses are a major motivation of the approach.

In this work we have presented an application of our approach to a program family with existing family members. To further validate the approach, it is also important to study its application to a program family which is not pre-existing. In this case, the abstract machine and DSL might be developed from the results of a domain analysis or a commonality analysis, such as FAST [11].

References

[1] B.R.T. 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, April 1995.

[2] J. Bentley. Programming pearls: Little languages. *Comm. of the ACM*, pages 711–716, August 1986.

[3] H.K. Berg, W.E. Boebert, W.R. Franta, and T.G. Moher. *Formal Methods of Program Verification and Specification.* Prentice-Hall, EngleWood Cliffs, NJ, 1982.

[4] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In *Coordination and models, Proceedings of the first international conference, Cesena, Italy*, number 1061 in LNCS, pages 75–88, 1996.

[5] 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.

[6] J. Bosch and G. Hedin, editors. *Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models, Linköping.* Technical Report 96-173, Lund University, April 1996.

[7] J. Graig Cleaveland. Building application generators. *IEEE Software*, July 1988.

[8] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [10], pages 54–72.

[9] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23^{rd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996.

[10] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in LNCS, February 1996.

[11] N.K. 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, January 1997.

[12] N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[13] N.D. Jones. What *not* to do when writing an interpreter for specialisation. In Danvy et al. [10], pages 216–237.

[14] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, EngleWood Cliffs, NJ, June 1993.

[15] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th IEEE International Conference on Software Engineering ICSE-18*, pages 542–553, 1996.

[16] David Ladd and Christopher Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, New Mexico, October 1994.

[17] G. Necula. Proof-carrying code. In *Conference Record of the 24^{th} Symposium on Principles Of Programming Languages*, pages 106–116, Paris, France, January 1997. ACM Press.

[18] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. http://www.sunlabs.com/people/john.ousterhout/, 1997.

[19] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for operating system specialization. In wDSL'97 [22].

[20] Scott Thibault and Charles Consel. A framework for application generator design. In *Proceedings of the Symposium on Software Reusability*, Boston, MA, USA, May 1997.

[21] A. van Deursen and P. Klint. Little languages: little maintenance? In wDSL'97 [22].

[22] *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997. Computer Science Technical Report, University of Illinois at Urbana-Champaign.

[23] The XFree86 Project. http://www.xfree86.org/.

A A Complete GAL Example

Appendix B gives a complete listing of the GAL specification for several models of S3 video adaptors. In this appendix, we explain some of the constructs that were not included in the main text.

Although the various registers of video cards are typically accessed using an addressing scheme, there is sometimes a sequential procedure that must be followed to access some registers. The `serial` construct is used to specify this kind of procedure (see listing). The construct consists of a list of sequences of actions that should be performed on the ports to access the registers. Thus multiple ports may be accessed during the procedure, as in the example. Each sequence consists of a port, an operation (`<=` write, `<=>` read/write, `=>` read), and a sequence of values for writes or registers names for reads and read/writes. The actions in the sequence are performed from the first port to the last, from left to right in the sequence. The mode (`R` read, `R/W` read/write, `W` write) to the right of the sequence indicates whether this sequence applies to reading the registers to writing the registers or both.

The serial construct in the example defines the registers `PLL1`, and `PLL2`. In order to write values to these registers the construct would be executed as follows. Write 3 to `misc[3..2]`, write the value of `PLL1` to `seq(0x12)`, write the value of `PLL2` to `seq(0x13)`, and finally, write 0, then 1, then 0 to `seq(0x15)[5]`.

The S3 specification also includes an example of a *derived field*, which is not discussed in the paper. This is a field whose value is derived from one of the standard fields. In the example, `StartFIFO` is a derived field. Its value is set whenever the graphics mode is set, and is based on the value of `HTotal`, the horizontal resolution. The declaration indicates this with the `from` clause.

The `clockmap` is used when a card has both fixed and programmable clocks such as the S3 Trio cards. It indicates which clocks are fixed and which are programmable. The example for the S3 indicates that clock 0 and 1 are fixed, clock 2 is not available (`NA`), and clock 3 is the programmable clock `f3`. The parameters `MinPClock` and `MaxPClock` are also related to clocks and specify the minimum and maximum values that can be generated by the clock (i.e. not all values of `f3M`, `f3N1`, and `f3N2` are valid).

Finally, the operating mode `access` is used to lock an unlock registers on the card.

B GAL S3 Listing

```
-- List all cards/models supported by this driver.
chipsets S3_911,S3_924,S3_80x,S3_928,S3_864,S3_964,S3_866,S3_868,
         S3_968,S3_TRIO32,S3_TRIO64;

-- Define ports.
port svga indexed:=0x3d4;
port seq indexed:=0x3c4;
port misc := 0x3cc, 0x3c2;

-- Define registers.
register Miscr:=misc;
register Slock:=seq(0x8);
register Offset:=svga(0x13);
register ExtChipID:=svga(0x2e);
register ChipID:=svga(0x30);
register Memory:=svga(0x31);
register State:=svga(0x36);
register Lock1:=svga(0x38);
register Lock2:=svga(0x39);
register StartFIFOr:=svga(0x3B);
register Misc1:=svga(0x3a);
register Control:=svga(0x42);
register Control2:=svga(0x51);
register HOverflow:=svga(0x5D);
```

```
register VOverflow:=svga(0x5E);
register Control3:=svga(0x69);
-- Serial registers (see appendix A).
serial begin
  misc[3..2]<=  (3,-   ,-,-,-) W;
  seq(0x12)<=>  (-,PLL1,-,-,-) R/W;
  seq(0x13)<=>  (-,PLL2,-,-,-) R/W;
  seq(0x15)[5]<=(-,-   ,0,1,0) W;
end;

-- Define predefined fields

-- Horizontal resolution fields.
field HTotal := HOverflow[0]#std;
field HEndDisplay := HOverflow[1]#std;
field HStartBlank := HOverflow[2]#std;
field HStartRetrace := HOverflow[4]#std;

-- Vertical resolution fields.
field VTotal := VOverflow[0]#std;
field VEndDisplay := VOverflow[1]#std;
field VStartBlank := VOverflow[2]#std;
field VStartRetrace := VOverflow[4]#std;

-- Virtual screen fields.
field LogicalWidth := Control2[5..4]#Offset scaled 8;
cases
for S3_928,S3_968,S3_TRIO32,S3_TRIO64
  field StartAddress := Control2[1..0]#Memory[5..4]#std;
for S3_80x
  field StartAddress := Control2[0]#Memory[5..4]#std;
for S3_864,S3_964
  field StartAddress := Control3[4..0]#std;
for others
  field StartAddress := Memory[5..4]#std;
end;

-- Define derived fields (see appendix A).
field StartFIFO from HTotal := HOverflow[6]#StartFIFOr offset 10 scaled 8;

-- Special S3 flags that must be set for 256 color graphics modes.
enable SVGAMode sequence is Misc1[4]<=1,Memory[3]<=1;

-- Define standard parameters.
param TwoBankRegisters:=false;
param InterlaceDivide := true;

cases
for S3_911,S3_924
  param RamSize:=State[5] mapped (0=>1024,1=>512);
for others
  param RamSize:=State[7..5] mapped (0=>4096,2=>3072,3=>8192,4=>2048,5=>5120,
6=>1024,7=>512);
```

```
end;

-- Define clocks.
cases
for S3_TRIO32,S3_TRIO64
  param NoClocks:=4;
  field ClockSelect:=Miscr[3..2];
  param MinPClock:=135;
  param MaxPClock:=270;
  field f3M:=PLL2[6..0] offset 2 range 1 to 127;
  field f3N1:=PLL1[4..0] offset 2 range 1 to 31;
  field f3N2:=PLL1[6..5] mapped (0=>1,1=>2,2=>4,3=>8);
  clock f3 is 14318*f3M / f3N1*f3N2;

  clockmap is (fixed,fixed,NA,f3);
for others
  param NoClocks:=16;
  field ClockSelect:=Control[3..0];
end;

-- Identification procedure.
identification begin
  1: ChipID[7..4] => (0x8=>step 2, 0x9=>S3_928, 0xA=>S3_80x, 0xB=>S3_928,
0xC=>S3_864, 0xD=>S3_964, 0xE=>step 3);
  2: ChipID[1..0] => (0x1=>S3_911,0x2=>S3_924);
  3: ExtChipID => (0x10=>S3_TRIO32, 0x11=>S3_TRIO64, 0x80=>S3_866,
0x90=>S3_868, 0xB0=>S3_968);
end;

-- Register locks on S3 chips.
enable access sequence is Lock1<=0x48,Lock2<=0xA5,Slock<=0x6;
disable access sequence is Lock1<=0x00,Lock2<=0x5A,Slock<=0x0;
```