Towards the Standardization of Plug-and-Play Devices for Model-Based Designs of Embedded Systems

Omair Rafique and Klaus Schneider
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract—We transfer the concept of plug-and-play devices from general purpose operating systems to sensors and actuators in model-based designs of embedded systems. So far, device vendors take the liberty of writing their device drivers for specific operating systems using their own vendor-specific implementation style. Consequently, the diversity of drivers limits the application of the device and the absence of standardization has even exposed faulty drivers leading to system crashes. In this paper, we therefore introduce the concept of the driver engine framework which generates drivers automatically from a specification provided by device vendors using our standard templates for a model-based design. Moreover, the generality is preserved by using an architecture description language which provides an abstract representation for interaction interfaces of devices.

I. INTRODUCTION

Model-based design flows try to systematically guide all phases of the overall system development process from the modeling phase to the final deployment phase. Such design flows start with abstract models and are supported by tool chains typically providing simulators, tools for verification, code generators, and tools for system and communication synthesis. These tools favor the reuse of already existing models and allow engineers to alter the system design even in late stages of the development process while still keeping tight time-to-market deadlines.

For this reason, model-based design flows are now widely used for the development of embedded systems, e.g., using Matlab/Simulink. This trend has also been adapted for real-time embedded systems, especially in the automotive domain. However, many embedded systems did not only become much larger during the last years; they also have to integrate now many more sensors and actuators. Especially in the domain of driver assistance systems, modern cars are now profoundly equipped with a large number of sensors. The manual integration of such devices by the end-user requires her/him to write additional code for accessing the sensors/actuators from the code generated by a model-based design tool, which can become a major obstacle to meet tight time-to-market deadlines.

We therefore envision the transfer of device drivers as a standardized concept for general purpose computers to the domain of model-based design of embedded systems. To this end, device vendors provide drivers for their devices to be triggered according to a specific model of computation [2]–[4], [6]. This implementation of the device driver is completely hidden from the end-user, and is instead made available as a library component in the model-based design to constitute a powerful Plug-and-Play (PnP) concept. However, the existing approach of PnP devices suffers from the generality, i.e., a missing standardization for models of computations used in model-based design. Instead, device vendors take the freedom of writing drivers on their own specific ways, i.e., using non-standard interfaces and trigger conditions.

We presented the idea of MoC drivers in [7]. A MoC driver wraps the actual sensor/actuator behavior in an abstract behavior that provides the MoC used by the model-based design. This paper continues with that idea and presents the work-in-progress namely the driver engine platform as a systematic tool for automatically generating those drivers for particular MoCs: We present the concept to stipulate a set of rules for device vendors in the form of standard templates. Using these templates, device vendors provide the specifications of their devices in a standardized format. Moreover, an architecture description language (ADL) is used to provide abstract representations of standard interaction interfaces for devices. The driver engine accepts both these standard specifications from the device vendors and the architecture description of standard interfaces, and generates the device drivers for the target hardware.

The outline of the paper is as follows: In Section II, the related work is presented. In Section III, we discuss the concept of existing PnP devices in general purpose computers and their limitations. In Section IV, we present our driver framework prior to the conclusions and discussion of future work in Section V.

II. RELATED WORK

Microsoft has developed an open source toolkit named .NET Gadgeteer1 [10] for building small electronic devices using the .NET Micro Framework. It combines the advantages of object-oriented programming, solderless assembly of electronics, and support for customizable physical design. The main idea conceived in this paper is to provide a standard mechanism to device vendors to automatically generate the device drivers while abstracting the behavioral details from the end user. Although it is not an embedded system and is merely restricted to defining the drivers for a specific development kit, we

envision a similar concept for the model-based design of embedded systems by abstracting the behavior of devices for various models of computation.

A general purpose computer running an operating system like Microsoft Windows, loads device drivers and invokes functions of the provided drivers to access the device [5]. Using dynamic loading of device drivers, a powerful PnP concept is obtained this way. However, in previous Windows versions, users suffered a lot from the Blue Screen of Death (BSOD). According to Microsoft, the main cause of BSOD were faulty device drivers, as written and provided by different vendors. A great success towards making the current Windows version much more stable has been achieved by static driver verification [1] that ensures a correct implementation of the driver. In contrast to this, we develop a correct-by-construction approach of the device drivers.

The Institute of Electrical and Electronics Engineers (IEEE) and Measurement Society’s Sensor Technology Technical Committee developed a set of smart transducer interface standards named IEEE-14512 [9]. The key element of these standards is the definition of the Transducer Electronic Data Sheet (TEDS). TEDS is a memory device which allows describing the identification, calibration, correction data, and manufacturer-related information for analog sensors. This concept is not widely adopted in model-based designs and is confined to analog devices. In contrast to this, our framework features analog as well as digital devices with the ability to generate drivers based on different MoCs in model-based designs.

The OS abstraction layer (OSAL) project provides the abstraction at the OS level, which isolates embedded software from the real-time operating system [8]. The OSAL provides implementations for several operating systems such as vxWorks and RTEMS. In addition to the OS APIs, this project also provides a hardware abstraction layer to provide portable interfaces to hardware devices such as memory, I/O ports, and non-volatile memory. It allows the development of portable embedded system software that is independent of a particular real-time operating system (RTOS). However, the strategy employed for this project assumes that the hardware is already abstracted with an operating system.

III. MODEL-BASED DESIGN FLOWS AND THE EXISTING PnP CONCEPT

Model-based design flows have been introduced to simplify the design process, to trace out specification discrepancies and design errors earlier in the development cycle, and to reduce the overall time-to-market. In real-time embedded systems, the interaction with the real world is an integral part of the design. This interaction with the surrounding system takes place by sensors and actuators. The attempt to manually integrate such devices deteriorates the idea of model-based design flows as it can possibly introduce implementation discrepancies (like BSOD) and inability to meet tight time-to-market deadlines.

Therefore, we envision to remove this shortcoming by introducing PnP devices in model-based designs of embedded
systems as known from general purpose computers. However, the existing concept of PnP devices expresses some limitations mainly from the perspective of how the corresponding device drivers are implemented. To this end, device vendors consider different implementation techniques for writing their drivers. Consequently, this leads to non-standardized code and fine-grained access limited to a particular operating system. A study presented in [5] shows that at least 44% of Linux drivers have code that is not captured by a class definition. Moreover 28% of drivers support more than one device per driver, and 15% of drivers perform significant computations over data. To this end, we observe a lack of generality and absence of standardization in the existing concept of PnP devices. Hence, we envision a standard mechanism for generating PnP devices automatically in model-based embedded systems. The key idea of this concept is to maintain standardization and to preserve generality while generating drivers.

IV. THE DRIVER ENGINE PLATFORM

The driver engine platform is organized in a systematic two-stage mechanism represented by the blue boxes shown in Figure 1. This platform is designed to comprehend and to execute the Device-and-Interface strategy as represented by the grey boxes in Figure 1. The term ‘device’ reveals the specification of the sensor as provided by the device vendor using our standard templates, whereas the term ‘interface’ is conceived as providing an abstract representation of standard communication interfaces. A graphical representation of the standard template for a temperature sensor is shown in Figure 2.

1) Device: The device part of this strategy is derived from a standard template. The taxonomy of these templates is based on the functionality of devices. For instance, different templates are constructed for different sensors capable of measuring parameters like temperature, light, humidity, motion, etc.. Templates are further classified on the basis of sensor types, e.g., analog or digital ones. Each template then requires specific parameters from the device vendors. We envision a domain-specific language to expose a common environment to device vendors for specifying the parameters of devices. To this end, the standard templates instruct device vendors on how to capture the specification of devices. A graphical representation of the template for a temperature sensor is shown in Figure 2.

The template is mainly composed of two parts: The first part acquires the general specification of the vendor. This involves parameters like name, family, type, range, and accuracy. For analog devices, the second part divulges the interface part of the sensor. The interface part is the mathematical representation that reveals the information about how the output (temperature in this case) is calculated for this device. The idea here is to avoid device vendors for writing even a single line of code for their drivers. Instead, the drivers are generated automatically based on the given information, thereby establishing a correctness-by-construction approach.

2) Interface: The interface part of this strategy employs an ADL with the aim to provide an abstract representation of standard communication interfaces. For instance, analog devices interact with embedded processors mainly by analog-to-digital converters (ADC). Similarly, for digital devices other standard interfaces are considered like SPI, I2C, USB, etc.. However, such interfaces are mapped at different addresses in different embedded processors. Therefore, the ADL here provides a generic representation of such interfaces. These are then mapped to the addresses as revealed by the memory map of the target processor. Therefore, the architecture description of the interface is provided independent of the hardware-specific implementation of the interface. Consequently, we address and overcome the absence of generality of PnP devices in model-based design as ignored in general purpose computers. The graphical representation of this acquired generality for ADC using our ADL is illustrated in Figure 3. The hardware-specific
addressing is simply attained by providing the addresses of the registers associated with specific interfaces. This then links the generic description of the interface to the provided API of the target hardware.

B. The Two-Stage Platform

The Device-and-Interface strategy is constructed to ensure standardization and to preserve generality. The idea is to expose a common environment to device vendors and hardware engineers to collect the specification of the device and the interaction model, respectively. The specification of the device is described using a domain-specific language whereas the architecture description for the communication interface is provided as a C program that is readable by the engine. The driver engine is composed of the following two stages:

1) Parser: Domain-specific languages (DSLs) are designed to be useful for a specific task in a fixed problem domain. We envision our standard templates to be exposed to device vendors as a DSL based on XML which is already used by various popular platforms: For example, Android applications are incomplete without the AndroidManifest.XML file, and similarly, .Net Gadgeteer expects a GadgeteerHardware.XML file which instructs the designer on how to show the module and the mainboard to the end user. The existence of many tools for parsing XML-tree structures makes it a useful candidate for standard templates. Moreover, the XML validator supplies a powerful mechanism for validating XML files against the XML schema. We intend to use the validator to check the templates provided by device vendors on conformance to our XML schema. This facilitates in tracing out faulty specifications earlier prior to driver generation.

To this end, the parser is envisioned as a tool for translating the device specification to a language that is readable and compatible with the driver engine. More specifically, the XML specification of a device is translated to a C file. This parsed file of the XML specification is then supplied to the generator stage. As shown in Figure 1, the template for the device DeviceA_SENSOR is parsed to a C file accompanied by a C structure. As discussed in Section IV-A, the first part of the C structure reveals the general specification whereas the second part accommodates the transfer function of the sensor. Practically, it is generated as a function pointer and is integrated as a member of the C structure.

2) Generator: The generator stage of the driver engine is supplied with the outputs of the device-and-interface stage. This corresponds to the parsed C file containing the specification of the device and the C file from the ADL containing the abstract representation of the communication interface, respectively. Additionally, this stage takes into account a driver skeleton as shown in Figure 1. The driver skeleton is an important building block of the platform as it provides the basis of the driver. It is further conceived as model of computation (MoC) dependent code skeleton of the driver. Therefore, different driver skeletons can be designed based on different MoC wrappers like data flow MoC, synchronous MoC, or event-driven MoC [2]–[4], [6].

The generator accepts the driver skeleton in the form of a C file and uses it as a base code. This base code is constructed with the ability to comprehend and execute the device-and-interface strategy. Therefore, it employs a generic base code that processes the C files as supplied by the parser and the ADL. In other words, the generator exploits the driver skeleton by providing the specification of the device and the information regarding the communication interface. Finally, the base code is supplied and filled with the required C structures yielding a PnP device driver. The generated device driver is then ready to be used by end-users by simply evoking the function call.

V. CONCLUSIONS AND FUTURE WORK

Model-based designs for embedded systems contribute in identifying subtle errors and modeling discrepancies early in the design, thereby reducing the overall time-to-market. However, in real-time embedded systems, the interaction with the environment takes place by incorporating a growing number of sensors and actuators.

Therefore, we envision a standard platform for automatic integration of sensors and actuators in the form of PnP devices to maintain and to preserve the characteristics of model-based designs. We envisage a platform that stipulates a common set of rules for device vendors, allowing them to specify the parameters of their devices in a standard format. Moreover, we integrate an ADL into the platform that allows us to describe the generic representation of standard communication interfaces. This simplifies the way of automatically integrating PnP devices in various target processors.

The future work will present our XML-based domain-specific language and the application of this concept for different sensors, as provided by different vendors.

REFERENCES