

Accepted Manuscript

DSML4DT: A domain-specific modeling language for device tree software

Sadik Arslan, Geylani Kardas

DOI: [10.1016/j.compind.2019.103179](https://doi.org/10.1016/j.compind.2019.103179)

To appear in: *Computers in Industry*

Published online: 19 December 2019

Please cite this article as: Sadik Arslan, Geylani Kardas, DSML4DT: A domain-specific modeling language for device tree software, *Computers in Industry*, doi: [10.1016/j.compind.2019.103179](https://doi.org/10.1016/j.compind.2019.103179).

This is a PDF file of an unedited manuscript that has been accepted for publication. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.



DSML4DT: A Domain-specific Modeling Language for Device Tree Software

Sadik Arslan^{1,2} and Geylani Kardas¹

¹International Computer Institute, Ege University, 35100, Bornova, Izmir, Turkey

²Kent Kart Ege Elektronik A.Ş., Research and Development Department, Izmir, Turkey

sadik.arslan@kentkart.com.tr, geylani.kardas@ege.edu.tr

Abstract

Device trees (DTs) provide description of devices and peripherals inside an embedded system with node specifications. However, developers mostly encounter difficulties in creating DT applications due to DT syntax different from the well-known general purpose programming languages. Moreover, both development and configuration of DT software components regarding different microprocessor architectures can be very hard and time-consuming for many embedded system developers. In order to eliminate these difficulties, we introduce a domain-specific modeling language, called DSML4DT, which provides the model-driven development (MDD) of DT software for the wide range of processor families. The evaluation of using DSML4DT was performed inside a company producing intelligent transportation systems. The comparative evaluation results showed that approximately 76% of DT structures can be obtained automatically only through modeling with DSML4DT. Comparing with the software development process currently followed in the company, the new MDD process reduced the time elapsed for implementing a DT software to half. Finally, feedbacks from the developers confirmed that they adopted the language particularly in terms of functional suitability, compatibility and reusability.

Keywords: Model-driven engineering, domain-specific modeling language, device trees

¹ Corresponding author. Tel: +90-232-3113223 Fax:+90-232-3887230.

1. Introduction

A Device Tree (DT) is a data structure that allows the identification of physical device components of embedded system hardware with nodes (Madieu, 2017). Configurations based on DTs can be applied during the development of embedded system software for various devices e.g. point of sales, mobile phones, automated fare collectors, network equipments, medical devices, bus driver terminals, vehicle trackers and home automation products. DTs are used within standards such as "Open Firmware" and "Power Architecture Platform Requirements", providing a complete technical description of hardware components (Devicetree Community, 2019). DT structure allows required peripheral operations to be performed without touching the core source code (e.g. (Devigne et al., 2017), (Li et al., 2018), (Jassi et al., 2018)). However, it is hard to implement DT structures, especially in the development of systems that operate in a large number of different microprocessor architectures due to need for repeating the same DT configuration.

Moreover, embedded system software developers may find it difficult to learn, prepare and use DT source files that are text-based and have a structure different from the syntax of existing programming languages (Arslan and Kardas, 2018). Each new platform-specific file to be used for DT implementations must be prepared separately and from scratch. In these files, blocks are coded in DT syntax according to hardware parameters used in the system. Additionally, the developer must know the microprocessor-specific hardware. Preparing these files is also challenging for developers who are familiar with the domain but have little or no knowledge and experience in software development. Moreover, coding and/or configuring DT components for different microprocessor architectures is time consuming.

We believe that applying a model-driven development (MDD) process including domain-specific modeling (Kelly and Tolvanen, 2008; Medini and Boucher, 2019), and using a domain-specific language (Kosar et al., 2016; Kosar et al., 2014; Nakamaru et al., 2019), may eliminate the abovementioned problems and facilitate the creation of DT applications. Hence, in this paper, we introduce a domain-specific modeling language (DSML), called DSML4DT, which provides the MDD of DT software for the wide range of processor families and kernels. In fact, DSML4DT is free of processor types and it can be used for creating DTs for any architecture conforming to the international "Devicetree" specifications (Devicetree Community, 2019). CPUs with both 32-bit and 64-bit addressing capabilities are supported by

DTs. To name a few, DTs can be prepared for the processors and architectures such as ARC, ARM, X86, PowerPC, Xtensa and also MicroBlaze soft processor; so it is possible to model DT components corresponding to all these processors and architectures with using DSML4DT.

Developers can visually model their software using DSML4DT's graphical syntax which conforms to the "Devicetree" specifications. DT models are validated inside the integrated development environment (IDE) of DSML4DT and artifacts, all required to implement the designed DT, are automatically generated via execution of the model-to-text transformations defined inside DSML4DT. DSML4DT's metamodel represents a platform-independent metamodel of DTs and hence instance DT models conforming to this metamodel can be converted to the DT configurations for different embedded system architectures. The related MDD process is described in this paper. In addition, a comparative evaluation of using this new language was performed inside one of the leading IT companies in Turkey which produces DT-based intelligent transportation systems and manufactures bus fleet terminals. We also discuss the results achieved from this evaluation.

The remainder of the paper is organized as follows: Section 2 gives the related work on developing DT software. Section 3 briefly discusses DT structure. Syntax and semantics definitions of DSML4DT are introduced in Section 4. A discussion of the MDD process supported with DSML4DT is given in Section 5. The evaluation of the language is presented in Section 6. Finally, we conclude the paper with Section 7.

2. Related Work

There are various recommendations in the literature for the generation of hardware driver code. For instance, Katayama et al. (2000) propose generating driver code for Unix-like systems. With using model-driven techniques, Chen et al. produce "Makefile" files for the Linux kernel (Chen et al. 2014). Similarly, a number of driver code is generated for a single platform in (King et al., 2012). Lecomte et al. (2011) describe rules to create UML models for embedded systems with applying MDD for a multiple-input–multiple-output process. However, all of these studies do not support DT configurations and generation of DT files.

Two recent studies (Neuendorffer, 2018; Jassi et al. 2018) consider the automatic achievement of DT configurations from embedded system designs. Neuendorffer (2018)

discusses the use of model-based approaches in the design of FPGA systems and states DTs can be created from system designs. Although the related design flow is exemplified through a case study, the automatic DT generation process is not given. Jassi et al. (2018) describe the use of GRIP tool, to facilitate the integration of hardware blocks defining Intellectual Property to System-on-Chips (SoCs). Like DSML4DT, GRIP is built on the Eclipse Modeling Framework (EMF) (The Eclipse Foundation, 2013) and both tools have similar features in terms of the creation of integrated circuit designs. Jassi et al. (2018) claim it is possible for their code generator to generate DT source files if their software is run on Linux operating system (OS), but no application example is found. In addition, the evaluations regarding the completeness of automatically generated code and the measurement of the speed of the software development process are not included. Finally, our DSML allows for the MDD of components for many additional interfaces such as USB, Serial Peripheral Interface (SPI), and Inter-Integrated Circuit (I2C).

When remaining DT software development studies are taken into consideration, the use of DT for creating virtual machine and peripheral component interconnect interfaces (Nikkel 2016; Devigne et al., 2017) and the logical structure of hardware configurations (Schüpbach et al., 2012) are encountered, but there is no automatic DT software production in these studies. A DT compiler is included inside “Altera SoC EDS” tool (RocketBoards, 2019). However, code generation is only possible for a single kernel version of Linux and does not have a general structure supporting different platforms. In addition to those efforts, both DTs and DT-based configurations are used in various purposes, e.g. multi-device support within BTFRS Linux file systems (Rodeh vd., 2013), support package optimization for microprocessors of some motherboards acting as a gateway in IoT networks (Gioia et al., 2016), preparation of driver terminals in public transportation (Arslan et al., 2017), management of interactive virtual hydroelectric generating equipment scenarios (Li et al., 2018), construction of embedded systems to be used in traffic sign identification (Farhat et al., 2019) and the production of real-time health monitoring devices (Swaroop, 2019). However, none of these studies follow MDD approaches during the implementation of required DTs.

3. DT Structure

Simply, a DT is a data structure that describes the configuration of a hardware. This structure contains information about the many parts of the embedded system, such as the processor,

memory, data paths and peripherals. The OS parses the DT structure during bootloading and determines how to configure the microprocessor and whole embedded system. The DT structure is also used to make decisions about device drivers to be installed.

The DT structure has a specific syntax starting with a node named root, represented by the “/” character. Multiple child nodes can be created from each parent node. The nodes can optionally include attribute values that contain additional data. The Device Tree Source (.dts) file format is used to express device trees and these files can be edited by software developers. The Device Tree Compiler Tool is used to convert DT descriptions in .dts format to the Binary Device Tree Blob (.dtb) format required by the OS.

In Figure 1, a DT structure fragment from the definitions for CPU nodes running in an ARM microprocessor system is shown as an example. The beginning of the DT root node is given in line 1. Each node definition is located between the braces “{” and “}”. The first child of the root node is the “cpus” node whose definition is written between lines 2-45. The architecture in here owns two processor cores, namely cpu0 and cpu1. Lines 6-37 include the definition of cpu0, the first child of the cpus node. The second child, cpu1 is defined between lines 39-44. For each processor core, information such as operating-points, clocks and clock-names need to be added separately and manually by the developers. In addition to know the details of this configuration, a developer should also deal with the specific syntax of these DT structures. It is worth indicating that Figure 1 includes a simple DT structure example which covers only the basic definitions of cpu components in a SoC configuration. A DT developer should also know and insert domain-specific configuration for other components such as memory units, communication channels and peripherals. An embedded system requires a large number of such information to be written manually in DT syntax when developing the corresponding DT software. Hence, the related development process is both time consuming and complicated for DT developers.

```

01 / { //The root Node
02     cpus { //Child of the root node
03         address-cells = <1>;
04         size-cells = <0>;
05
06         cpu0: cpu@0 { //First child of the cpus node
07             compatible = "arm,cortex-a9";
08             device_type = "cpu";
09             reg = <0>;
10             next-level-cache = <&L2>;
11             operating-points = <
12                 /* kHz  uV */
13                 996000 1275000
14                 792000 1175000
15                 396000 1150000
16             >;
17             fsl,soc-operating-points = <
18                 /* ARM kHz  SOC-PU uV */
19                 996000 1175000
20                 792000 1175000
21                 396000 1175000
22             >;
23             clock-latency = <61036>; /* two CLK32 periods */
24             clocks = <&clks IMX6QDL_CLK_ARM>,
25                 <&clks IMX6QDL_CLK_PLL2_PFD2_396M>,
26                 <&clks IMX6QDL_CLK_STEP>,
27                 <&clks IMX6QDL_CLK_PLL1_SW>,
28                 <&clks IMX6QDL_CLK_PLL1_SYS>,
29                 <&clks IMX6QDL_PLL1_BYPASS>,
30                 <&clks IMX6QDL_CLK_PLL1>,
31                 <&clks IMX6QDL_PLL1_BYPASS_SRC>;
32             clock-names = "arm", "pll2_pfd2_396m", "step",
33                 "pll1_sw", "pll1_sys", "pll1_bypass", "pll1", "pll1_bypass_src";
34             arm-supply = <&reg_arm>;
35             pu-supply = <&reg_pu>;
36             soc-supply = <&reg_soc>;
37         };
38
39         cpu1: cpu@1 { //Second child of the cpus node
40             compatible = "arm,cortex-a9";
41             device_type = "cpu";
42             reg = <1>;
43             next-level-cache = <&L2>;
44         };
45     };

```

Figure 1: An example of DT structure

4. Syntax and Semantics of DSML4DT

The abstract syntax of DSML4DT language is defined with a metamodel developed according to DT specifications given in (Devicetree Community, 2019). The preliminary version of the metamodel is introduced in (Arslan and Kardas, 2018). This initial metamodel has been revised and extended in this study to complete the component relations required especially on SoC and peripheral DT descriptions. The metamodel is divided into five different viewpoints, called *Core*, *SoC*, *Aips_Bus*, *Spba_Bus* and *Peripheral*. The placement of the meta-entities inside these viewpoints is made according to DT usage in embedded systems. The metamodel

is encoded with Eclipse Ecore (The Eclipse Foundation, 2013) and hence it is possible to integrate the metamodel with various MDD tools based on EMF.

DSML4DT's metamodel is composed of more than 70 meta-entities (corresponding to DT components) and their relations. Due to space limitations of the journal, it is not possible to discuss all meta-entities, and in here we only give the brief descriptions of the viewpoints. Similarly, the partial Ecore diagram illustrating the Core viewpoint is shown only. However, the whole metamodel, Ecore diagrams for all viewpoints and the complete specification of all DSML4DT meta-entities and their relations can be found in the accompanying Mendeley data repository (Dataset, 2019).

In the following, five viewpoints of DSML4DT (Core, SoC, Aips_Bus, Spba_Bus and Peripheral) are briefly described. Elements and associations covered in each viewpoint are indicated in the text with italics.

Core Viewpoint: Figure 2 shows the Ecore representation for this viewpoint. Element *root* is the base node from which the entire system is produced. The root element is in “has-a” relationship with other elements derived from the root. The processor and memory of the embedded system are defined in this viewpoint. For multi-core processors, the necessary derivations for each core unit are made from the *cpus* element. In addition, there are aliases and chosen elements which do not have any hardware relations, and these elements own abbreviations, definitions, and boot parameter transitions to be used in all DT structures.

SoC Viewpoint: This part of the metamodel describes the SoC integrated circuits. Parameter settings of many SoC features such as sound, image, and timer are provided inside this viewpoint. Moreover, the *interrupt_controller* element handles the generation and the adjustment of all interrupts in the embedded system.

Aips_Bus Viewpoint: SoC peripheral components with low bandwidth communicate with SoC units via Advance High Performance Bus to Internet Protocol (Aips) Bus interface in embedded systems. Within Aips_Bus viewpoint, it is possible to model DT elements such as *caam*, *iomuxc*, *ldb*, *usb*, *fec*, *i2c*, *uart*, *pwm*, *flexcan*, *gpio*, *wdog*, *clks*, *usbphy* and *spba_bus*.

Spba_Bus Viewpoint: In embedded systems, Shared Peripherals Bus Interface (SBPA) Bus is used for communicating with some shared external units. This interface enables the communication between the Smart Direct Memory Access core and the peripherals. Sbpa_Bus viewpoint supports modeling DT structures such as *spdif*, *esai*, *ssi* and *ecspi*.

Peripheral Viewpoint: Hardware units, which are not on SoC integrated circuit and connected to SoC from the outside, are modeled in DSML4DT according to this viewpoint. Integrated circuit peripherals with different audio and video codecs are defined. Peripheral elements like *clocks*, *battery*, *gpio_keys*, *sound*, *sound_spdif*, *sound_hdmi*, *lcd* and *backlight* are generated directly from the root element in the metamodel.

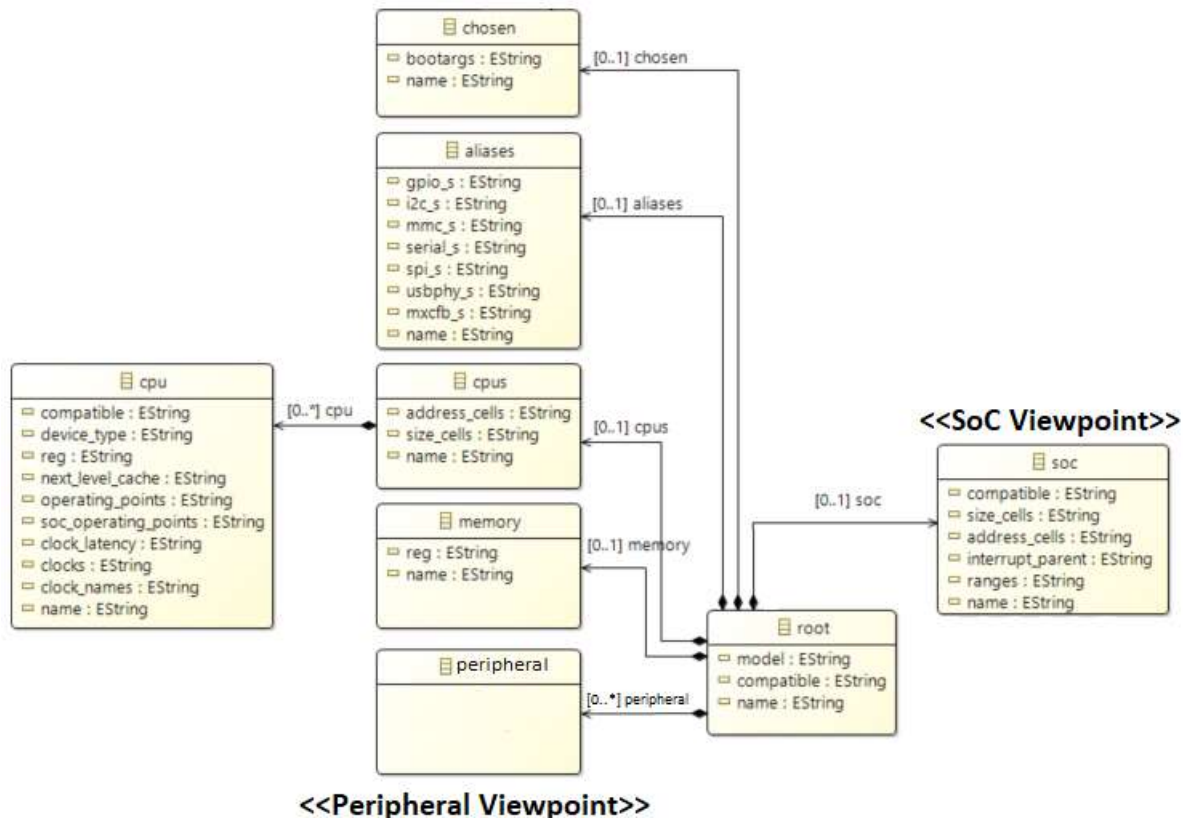
























Figure 2: DSML4DT's Core Viewpoint

We also provide a graphical concrete syntax which maps DSML4DT's abstract syntax elements to their graphical notations. In order to construct DSML4DT's concrete syntax, we benefited from the features of Sirius (The Sirius Project, 2016) modeling environment. Both providing a tool for implementing a graphical editor from an Ecore metamodel and allowing one to define dedicated editors including diagrams based on a viewpoint approach caused us to build DSML4DT's graphical modeling toolset on Sirius environment.

Graphical notations for the abstract syntax meta-elements were determined first and tied to the domain concepts with using Sirius. Table 1 lists some important notations. A screenshot from Sirius-based IDE of DSML4DT is given in Figure 3. It is possible to create model diagrams

for each viewpoint (see left upper section in Figure 3). Model symbols, associations and names can be seen in these diagrams. All editors for the viewpoints have a palette (seen at the upper right of Figure 3). In here, the items specific for the concrete syntax of each DSML4DT viewpoint are listed. Hence, developers can create DT models conforming to DSML4DT specifications by simply drag-and-dropping required items from the palette.

Table 1: Some of the concepts and their notations provided for DSML4DT's graphical concrete syntax

Concept	Notation	Concept	Notation
Root		Timer	
Cpu		L2 Cache	
Memory		Pwm	
SoC		Flexcan	
Interrupt Controller		Gpio	
Aips Bus		Wdog	
Ipu		Spdif	
Hdmi Core		Esai	
Hdmi Cec		Uart	
Power		Lvds Channel	
Sound		Battery	

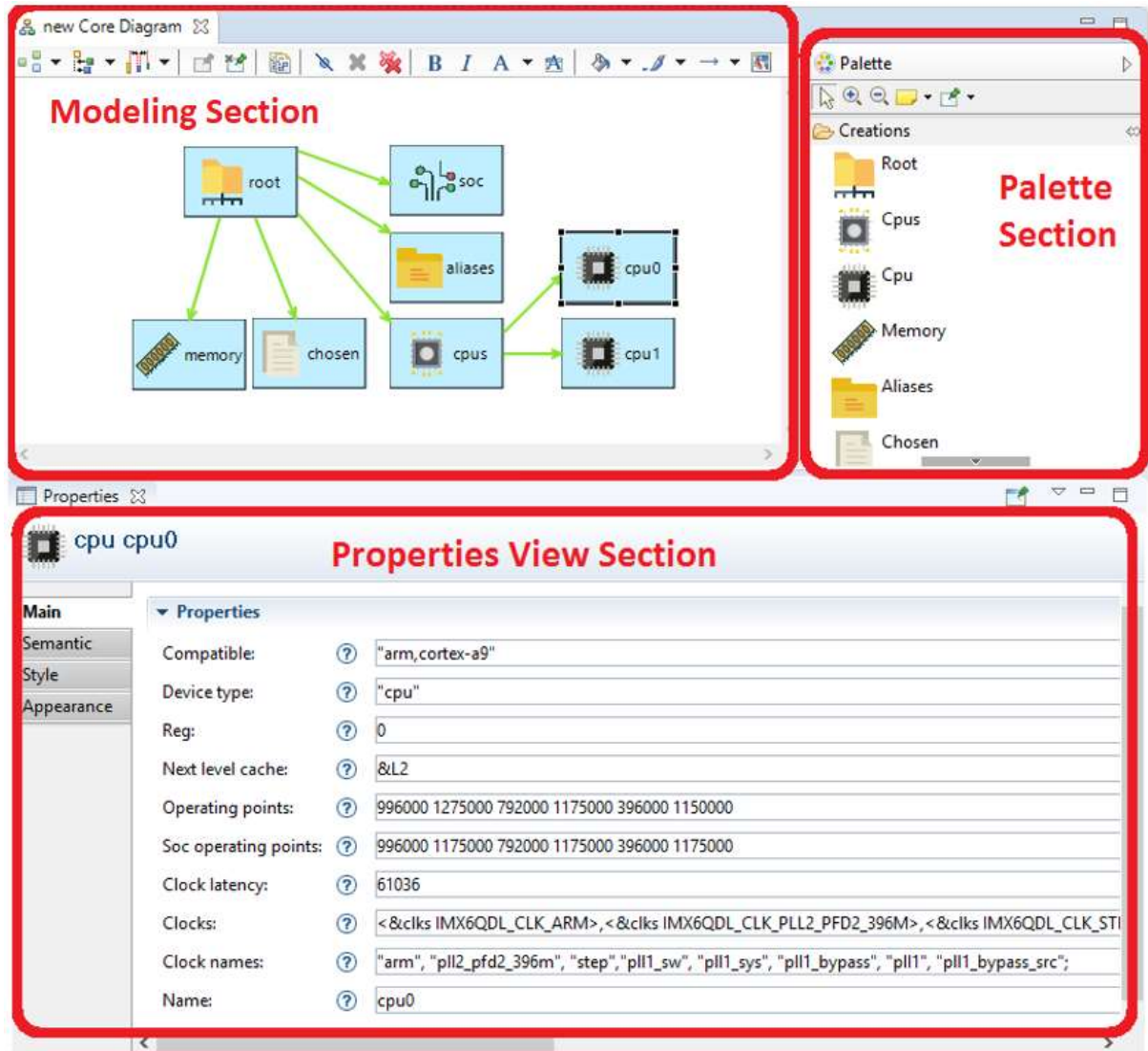


Figure 3: A screenshot from DSML4DT IDE

Palettes for the viewpoints have modeling elements specific for the related DSML4DT viewpoint. For example, the palette section seen in Figure 3 currently includes modeling elements only specific for the Core Viewpoint. However, some common modeling elements can be found in different viewpoints. For example, the SoC element is both located in the Core and SoC viewpoints. For this reason, a SoC node created in the Core viewpoint diagram is automatically added to the SoC diagram. Thus, the consistency between the viewpoints of the DT models is ensured. Such features are provided by the static semantics of DSML4DT. Finally, at the bottom of the editor, a Properties section resides. In this section, all remaining hardware properties pertaining for each modeled device element can be entered. Hence, all model properties can be completed via this development tool. In Figure 3, this section currently shows how the properties are entered for the selected model element (one cpu

instance, called *cpu0*) in the diagram. To insert values for the properties of another DT model element, this element located at the Modeling section can be clicked and hence its properties are listed in the Properties section. DSML4DT IDE provides the interface for the completion of all model elements' properties defined according to Devicetree standard which leads to completely generate DT definitions for the corresponding DSML4DT models. However, some properties depend on the different types of peripheral integrated circuits and their definitions may not be supported in the Devicetree standard, hence they can not be fully modeled with DSML4DT. Such properties of these peripherals need to be added manually into the generated definitions. For instance, let us consider an integrated circuit increasing the number of GPIOs. This circuit works with SPI interface. Figure 4 shows the DT node description prepared for this circuit. Code lines between 1-5 and 9-13 are auto-generated by DSML4DT. However, the standard DT structure does not support some additional device properties required for this circuit (shown in bold in Figure 4 between lines 6-8) and both these non-standard properties and the corresponding values should be manually added to the configuration.

```

01      gpiom1: gpio@0 {
02          compatible = "microchip,mcp23s08";
03          gpio-cells = <1>;
04          gpio-controller;
05          reg = <0>;
06          mcp,spi-present-mask = <0x03>;
07          mcp,gpio-base = <300>;
08          mcp,spi-max-frequency = <10000000>;
09          interrupt-controller;
10          interrupt-parent = <&gpio6>;
11          interrupts = <11 4>;
12          interrupt-cells = <2>;
13      };

```

Figure 4: An example of DT structure in which some parts are auto-generated by modeling with DSML4DT while remaining is manually added to complete the configuration.

Constraint-checks and static semantics controls are automatically made inside the environment according to DSML4DT model validation rules. These rules were written with using Acceleo Query Language (AQL) in the Sirius platform (AQL, 2018). EMF models can be queried with AQL. Moreover, rules written with AQL bring strong model validation including type checking at the validation time. AQL interpreter is used in Sirius to execute written queries (validation rules) on EMF-compliant system models. For each DSML4DT

viewpoint, we created validation rules in AQL syntax and they are ready to be used during MDD of DTs. For instance, the following AQL rule checks whether at least one *power* attribute is supplied for a *gpio_keys* instance inside a DSML4DT Peripheral viewpoint:

```
aql: self.power-> notEmpty ()
```

Following the “aql” command the rule starts after “:”. “self.power” uses the power attribute value in the *gpio_keys* instance element. Then “->” specifies that this attribute value cannot be empty by applying “notEmpty()” query. Similarly, execution of the following AQL rule confirms the naming of each *memory* instance inside a DSML4DT Core viewpoint:

```
aql: self.name-> one (str | str.equals ('memory'))
```

In fact, developers do not need to know both the definition and the structure of these rules since the rules are automatically applied on a DSML4DT instance model without any user intervention and error messages are shown to the user inside the IDE when the model validation fails.

The rules enable checking constraints such as compartment (e.g. DT *memory* and *battery* elements can be created only from the *root*), number of relationships between DT model elements (e.g. only one *cpus* element can be derived from *root*) and source and destination elements in a relationship (e.g. *battery* can be created from the *root* but not vice versa). Moreover, the constraint-checks also include some editorial features which assist the user in the design process while creating the model, such as transition between viewpoints (e.g. design *SoC* before *Core* model), unification for all elements (e.g. a *root* element created inside *Core* is automatically added into *Peripheral* model), integrity of relationships and elements inside all viewpoints (e.g. when a DT element is deleted, all its relations inside all DSML4DT viewpoints are deleted automatically).

Moreover, validation of the DT models according to DSML4DT static semantics specifications is also performed in the IDE. Inclusion of the mandatory elements (e.g. Model Validation Rule: *Peripheral_1* - *At least one key element must be generated from the gpio_keys*) and cardinalities of the elements in the placeholders (e.g. Model Validation Rule: *Aips_bus_6* - *One display_timings element must be in the aips_bus diagram*) are controlled. Finally naming conventions are also validated (e.g. Model Validation Rule: *Core_10* - *The memory element should be named as “memory”*). Additional examples of these validation

rules are available in (Dataset, 2019). Use of these rules and appearance of the validation messages are exemplified in the next section.

In order to generate executable DT files from DSML4DT models, we defined a series of model-to-text (M2T) transformation rules by using Acceleo (Acceleo, 2018). Acceleo provides a tool as an Eclipse plug-in where M2T transformations can be written, parsed, checked, and directly executed inside the IDE. It enables the definition of code generation rules and also supports the interpretation of these rules. The semantics of DSML4DT language is provided over the application of these rules at run time on DT models conforming to DSML4DT syntax. Hence, the generation of DT source code from DSML4DT models is possible. The generated code for DTs can be directly executed within the embedded OS environment. Examples of the written M2T rules can be found in (Dataset, 2019).

5. Model-driven DT Development Methodology based on DSML4DT

Features of DSML4DT language discussed in Section 4 can be used to constitute a model-driven DT development methodology in which developers can visually design and implement their DT-based embedded systems. The proposed model-driven methodology includes system modeling and automatic code generation for exact DT implementations.

In the system modeling step, a developer uses the fully functional graphical IDE of DSML4DT to design the system-to-be-developed. DSML4DT's concrete syntax covers 5 viewpoints and hence for each viewpoint, a modeling palette is provided. The tool does not only offer a computer-aided design for system modeling, but also supports various automatic constraint checks and semantics controls via model validation which lead the designers to create accurate models. The main outcome of this step is DT software models conforming to DSML4DT specifications.

The next step is the code generation and completion for DT software. The output of the previous step will be the input for the execution of this step. In here, DT models are converted into DT code for the targeted embedded system. The M2T transformation rules are automatically executed on the DT instance models and code is obtained for the DT implementation. The developer does not need to know about both the context of M2T rules and their execution details. (S)he only selects the code generation feature over the IDE. The

result of the automatic code generation step is DT source files. These files can be used later in the OS where they are required.

It is worth indicating that DT code is generated according to DSML4DT instance models created by the DT developers. In fact, each instance model consists of more than CPUs and memories, i.e. it is possible to model a SoC with many sound and video units, communication interfaces and additional peripherals. In addition to the visual design of a SoC, DSML4DT IDE enables the inclusion of additional properties specific for the related integrated circuits. DSML4DT syntax is based on the international “Devicetree” specifications (Devicetree Community, 2019) and abstract from a specific processor or a hardware system. As discussed in the introduction, currently it is possible to produce hardware configurations for various processors and architectures with using Devicetree specifications; hence DSML4DT supports the MDD of DTs pertaining to all these processors. Same is valid for future processors supporting Devicetree specifications. DT developers can model SoCs including these new processors again with DSML4DT and DT code for these new hardware can automatically be achieved from these instance models.

In the following subsection, development of a DT software for a dual core embedded system is discussed in order to provide some flavor of using above MDD methodology.

5.1. Development of a DT software for a bus driver terminal

The device, for which a DT software is required, is a driver terminal (computer) used in public transportation buses. This device can be supported with Android or Linux OS where DT structure can be used. The device is used by the bus drivers and it can perform operations such as reading RF cards, monitoring environment temperature and making voice announcements, e.g. inside a bus. The hardware of this driver terminal (Figure 5) is designed and manufactured by Kentkart Co. (Kentkart, 2019).



Figure 5: The driver terminal device

The driver terminal device is a computer having an i.MX6 Dual Lite series micro-controller with ARM dual core CPU. The device also has 1 GB RAM and 4 GB storage unit. Modules such as SPI interfaced memory, Secure Digital interfaced MMC, gigabit Ethernet, Real Time Clock and USB interface are all available for the multimedia system support. A power management controller is used in the device. The ignition and odometer signals of the vehicle can be used in the device. Hard disk support of the system is provided by the SATA interface. In order to access the CAN in the vehicles, the microprocessor CAN interface is used. There are also camera interfaces and an audio converter integrated circuit that can work with I2S interface for audio output. 24-bit LCD is supported with RGB or LVDS interfaces, and LCD backlight supply circuit is located in the device. There is an RF card read/write circuitry and the Secure Access Module is available for safe RF access. Accelerometer sensor, EEPROM, temperature sensor and digital potentiometer are also included. These units are controlled by the I2C interface. There is a buzzer circuit for warnings. Finally, the device has GPS, GSM and Wi-Fi modules.

For this device, all hardware parts, briefly introduced above, need to be described in the DT layer of the embedded system architecture. Its DT software is basically located between hardware and OS drivers and DT files are compiled independently from OS kernels. All hardware parameters of the terminal pass through from DT binaries to OS kernels during the boot up stage of the embedded system.

In this example, DTs for the driver terminal described above are generated with using DSML4DT. Conforming to the proposed MDD methodology, we start by creating system models according to 5 viewpoints of DSML4DT. For instance, Figure 3 shows the Core

model designed for this hardware. First, a DT *root* node instance is created by drag-and-dropping, and then *memory*, *chosen*, and *aliases* nodes, mandatory for DT structure, are added. The driver computer has a dual core processor, so 2 *cpu* nodes are created. In addition, *SoC* node from which SoC features will be generated is created here. Hardware dependent properties of all nodes are also entered during this stage.

For another example, Figure 6 shows the *Aips_Bus* model designed for the driver terminal. Many features of the device and support of different interfaces are provided within this viewpoint. There are 2 *Aips_Bus* lines in the device. These lines are modeled with *aips1* and *aips2* nodes. 7 *gpio*, 4 *pwm*, 2 *flexcan*, 2 *usbphy* and 2 *wdog* nodes are produced from the *aips1* node. Also *clks*, *iomuxc*, *ldb* and *spba_bus* nodes are created from *aips1*. Remaining SoC, *Spba_Bus* and Peripheral models created for the specifications of the driver terminal can be found in (Dataset, 2019) with their discussions.

The validation of the designed model is performed inside IDE according to the constraints and rules previously discussed in Section 4. During this automatic validation process, error messages are shown to the developer if there are any violations. Figure 7 illustrates how developers are notified if some errors are encountered during the validation of a Core model.

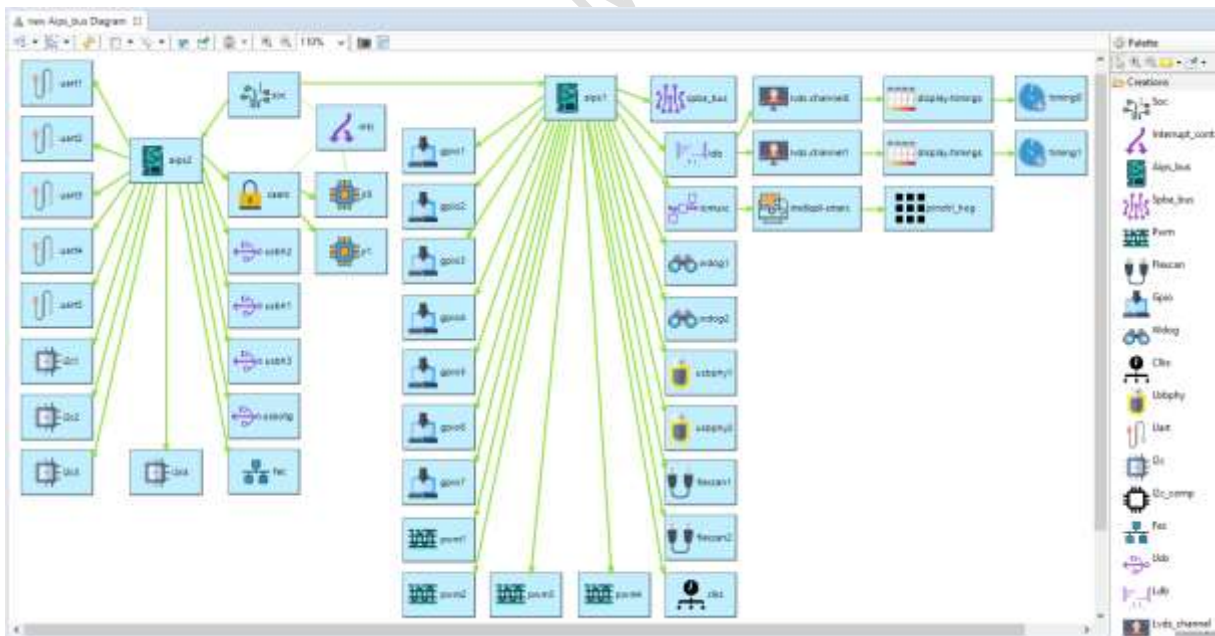


Figure 6: *Aips_Bus* model of the driver terminal designed inside DSML4DT IDE

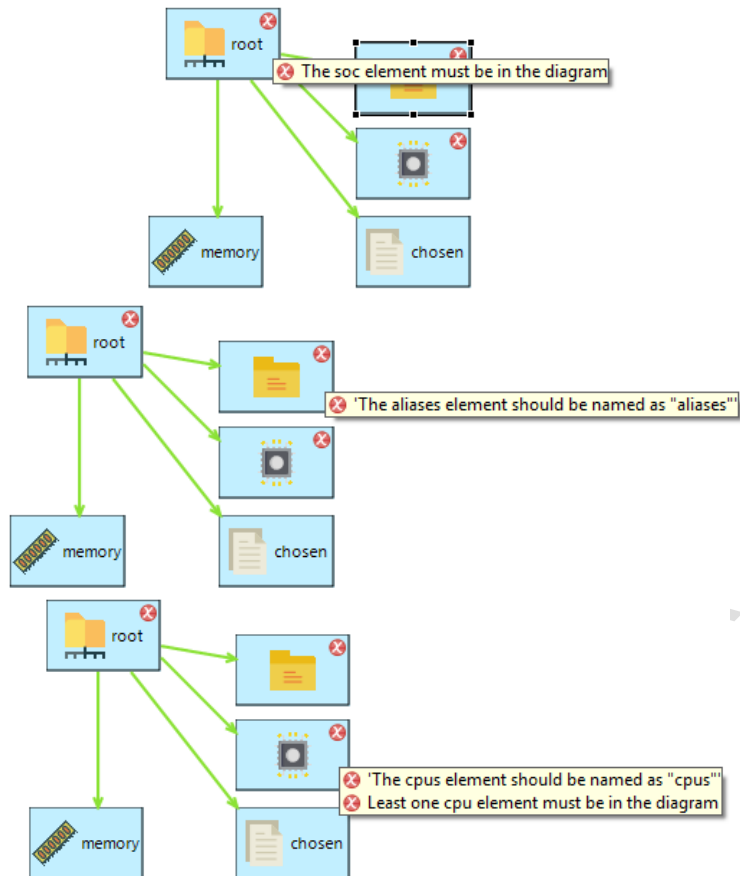


Figure 7: Some validation examples for the driver terminal DT model

After the design and validation of models are completed, DSML4DT's code generator works on these models and hence produces DT code ready to be executed. Figure 8 includes a fragment from the auto-generated DT code for the device terminal. M2T transformations are applied on the Core model of the driver terminal for root, cpus and cpu instances. Lines 1-3 in Figure 8 show the generated code for the root DT node. DSML4DT's code generator determines each root instance and the corresponding "model" and "compatible" attributes and then produces the required code in DT syntax. Similarly, code for *cpus*, *cpu0* and *cpu1* DT elements are generated between lines 5-7, lines 9-19 and lines 21-26 respectively. For this case study, a total of 906 lines of executable DT code is automatically generated from the DSML4DT models designed for the driver terminal. Whole code is available again in (Dataset, 2019).

```

01    /{
02        model = Kentkart i.MX6 DualLite Smart Device Board;
03        compatible = "fsl,imx6dl-sabresd", "fsl,imx6dl";
04
05        cpus{
06            address-cells = <1>;
07            size-cells = <0>;
08
09            cpu0{
10                compatible = "arm,cortex-a9";
11                device_type = "cpu";
12                reg = <0>;
13                next-level-cache = <&L2>;
14                operating-points = <996000 1275000 792000 1175000 396000 1150000>;
15                fsl,soc-operating-points = <996000 1175000 792000 1175000 396000 1175000>;
16                clock-latency = <61036>;
17                clocks = <&clks IMX6QDL_CLK_ARM>,<&clks IMX6QDL_CLK_PLL2_PFD2_396M>,
18                    <&clks IMX6QDL_CLK_STEP>,<&clks IMX6QDL_CLK_PLL1_SW>,
19                    <&clks IMX6QDL_CLK_PLL1_SYS>,<&clks IMX6QDL_PLL1_BYPASS>,
20                    <&clks IMX6QDL_CLK_PLL1>,<&clks IMX6QDL_PLL1_BYPASS_SRC>;
21                clock-names = "arm", "pll2_pfd2_396m", "step","pll1_sw", "pll1_sys",
22                    "pll1_bypass", "pll1", "pll1_bypass_src";
23            };
24
25            cpu1{
26                compatible = "arm,cortex-a9";
27                device_type = "cpu";
28                reg = <1>;
29                next-level-cache = <&L2>;
30            };
31        };
32    };

```

Figure 8: An excerpt from the generated DT file

After the DT file is produced, it is ready for processing in an OS kernel. DT file is copied to a directory in the kernel. This directory is predefined and fixed in the kernels. The DT software stored in this file is in .dts format and text-based. Then, this .dts file is compiled by the DT compiler and a .dtb file is created in binary format as previously discussed in Section 2 of this paper. That file is now suitable for the execution in the kernel. When the OS is initialized, the .dtb file is parsed and the necessary hardware parameters are taken from this file. Hardware components (CPU, memory, peripherals, etc.), whose descriptions are received from this file, can now be used and managed by the kernel. Based on the current DSML4DT implementations, we see that creating DTs with DSML4DT does not affect both the way and the speed of processing and managing these files inside OS kernels.

6. Evaluation

In order to determine the feasibility of using DSML4DT for the development of DT software, a comparative evaluation was performed. For this evaluation, we adopted the evaluation framework proposed in (Challenger et al., 2016) which enables the assessment of language constructs and the use of DSMLs according to various dimensions and criteria. The scope of our evaluation here covers Development Sub-dimension (under Execution Dimension) and User Perspective Sub-dimension (under Quality Dimension) of this framework. Therefore, the evaluation criteria pertaining to these dimensions, called Output Performance (Generation Performance), Development Time Performance, and Qualitative Analysis by a questionnaire, are taken into consideration. We revised these dimensions and criteria to make them meaningful and relevant for evaluating DSML4DT. We aimed at finding answers to the following research questions (RQs)?

RQ1: To what extent does the use of DSML4DT allow the automatic generation of DT components?

RQ2: Does the use of DSML4DT reduce the DT development time?

RQ3: What are the pros and cons of using DSML4DT from DT developers' perspective?

To find answers for the above RQs, our evaluation consists of two parts: 1) quantitative analysis, including generation performance and development time evaluations 2) qualitative assessment within user perspective.

The whole evaluation was carried out in Kentkart Ege Elektronik Company (shortly Kentkart) (Kentkart, 2019). Kentkart is one of the leading IT companies in Turkey which produces various intelligent transformation system solutions for the automated fare collection, vehicle tracking, real-time passenger information, route planning and on-board video surveillance. Currently, mass-transit systems of Kentkart are being used in more than 25 cities of Turkey and more than 10 worldwide locations in countries including Hungary, Macedonia, Pakistan, Poland, Serbia, United Arab Emirates and United States. Kentkart also manufactures hardware such as ticket vending machines, turnstile validators and bus driver terminals. Huge amount of software for all these devices and information systems are DT-based and/or configured with DT structures.

Five software developers working in the R&D center of the company were voluntarily participated in our study as the evaluators. These developers were first asked to develop DT software for four different systems by using their conventional DT development approach followed inside the company. Later, they were asked to apply the model-driven DT development methodology covering the use of DSML4DT to develop the same systems. All evaluators have at least a B.Sc. degree in computer / electrical engineering. Two of them also have M.Sc. degree in electrical engineering and pursuing Ph.D. in information technologies at the time of this study conducted. All of the evaluators are experts in embedded software development with varying experience from 5 to 11 years. Specifically, they have an average of 4 years of experience on design and implementation of DT-based systems. None of them previously used MDD techniques during DT implementations.

The evaluation process has the following stages: First, all evaluators received an MDD review. This step ensures countervailing their level of familiarity to software modeling. All evaluators received an introduction to DSML4DT language and its IDE for modelling and code generation. Then, a briefing on the case studies consisting of developing different DT systems was given to evaluators. Finally, the evaluators developed the required DTs by going through steps of analyzing use cases, designing/modelling systems, implementing/generating code and testing. Elapsed times were recorded for each developer and each step. In addition, artifacts of all experiments were evaluated to find out especially the generation performance of DSML4DT.

DT software development for four different systems with varying complexities was taken into consideration as the use cases. One of these systems was the bus driver terminal previously discussed in Section 5. Remaining DT-based systems were required to be developed and installed on three different hardware all manufactured by Kentkart. These are briefly described below:

Network Video Recorder (NVR): This device includes an i.MX53 series single core processor, 1 GB RAM, 512 MB NAND Flash and network units. This device does not have any peripheral that plays video or audio files. There is no LCD and touchscreen interfaces for the user interaction. The device has Linux OS.

Validator: This device is used in fare collection systems for public transportation. Usually RF cards are used to collect fees on public transportation via this device. It includes an Intel PXA270 series single core processor, 256 MB RAM, 128 MB NAND Flash and network

units. Unlike NVR, the validator has additional features such as RF card read/write and LCD support. It has Linux OS.

Multimedia Player: This device meets multimedia needs of public transportation. Basically, the device has LCDs via HDMI interface and provides audio output for announcements in buses. It also supports SATA interface, GSM, GPS and owns serial ports. The device includes an i.MX6 Quad series quad core processor, 1 GB RAM and 8 GB EMMC storage. There is optional LCD support via LVDS interface. It works on Android OS.

6.1. Results and Discussion

6.1.1. Quantitative Analysis

This analysis consists of measuring the generation of artifacts (called generation performance), and the time saved during whole development process (called development time performance).

Generation Performance:

DSML4DT's generation performance was calculated to answer RQ1. For this purpose, the comparisons between the automatically generated code and delta code added to complete the system are made. All evaluators developed their systems required for each case study with using DSML4DT to generate DT code and then completed this code to build the DT system. Performance evaluation is fulfilled by comparing the percentage of artifacts automatically generated and manually developed. These artifacts are Lines of Code (LoC) of DT files required for the systems namely, Driver Terminal, Multimedia Player, NVR and Validator devices. Measurement results are shown together in Figure 9. The LoCs listed for each case study is the average of all developers in the evaluator group. For instance, all evaluators obtained auto-generated code depending on their DSML4DT models for the driver terminal case study with varying ratios. 76.31% LoC is their average. The evaluators needed to manually add %23.69 LoC on average to complete their driver terminal DT software. The Overall Average is the LoC average obtained from all case studies. All measurement results for each developer and for each case study can be found in (Dataset, 2019).

According to the results in Figure 9, the average generated LoC rate is 75.52% for the whole evaluation. Generated LoC varies between approximately 71% and 83% for different case

studies. The most important reason for these variations are the device architectures used in the case studies. For example, Validator device has many DT elements whose modeling are directly supported by DSML4DT, and hence 83% of its DT software is automatically generated. However, in the architecture of the NVR device, there exist many software components apart from DT elements, hence the average LoC decreases to 71%. In addition, both the developers' knowledge and experience on each device and the quality of models created by the developers naturally have effect on the generated LoCs. In order to keep this effect minimum, we conducted this multi-case evaluation with varying DT development complexities instead of only one use case. Thus, a rate of 75.52% LoCs for the auto-generation with just using DSML4DT is quite high taking into consideration the variations encountered both in the needs of the DT applications for each case study and the hardware architectures of used embedded systems.

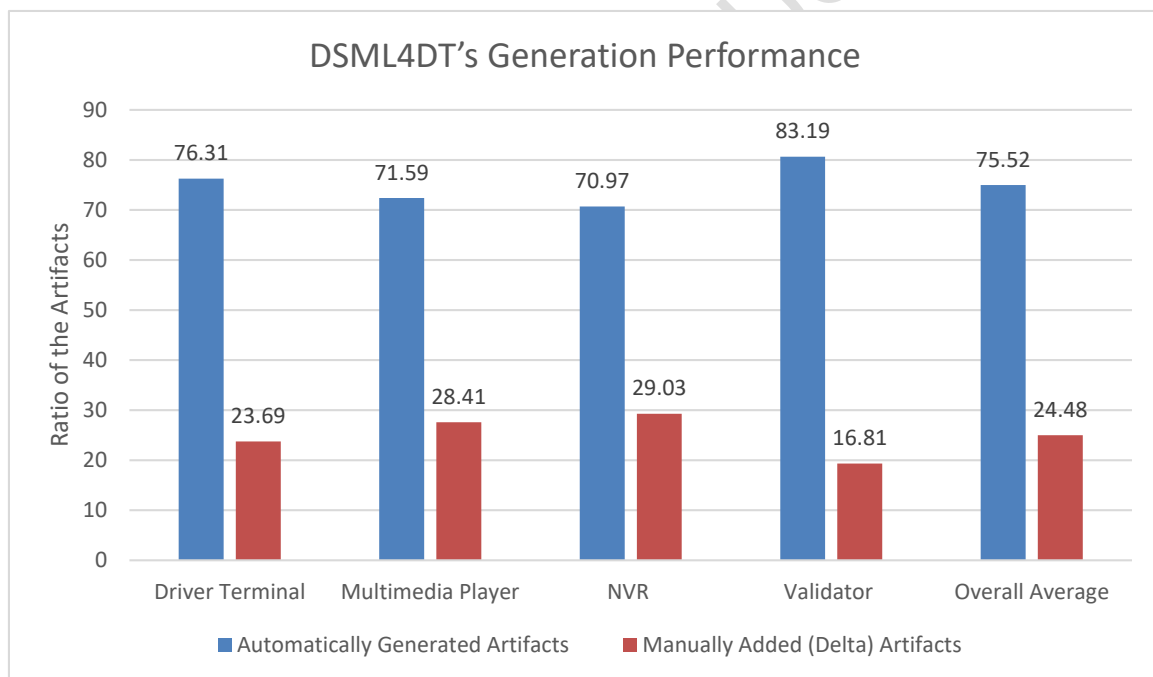


Figure 9: DSML4DT's generation performance

Development Time Performance:

In order to answer RQ2, we compared and analyzed the times which were recorded during Kentkart engineers developed the requested DT systems with using the new MDD process (Experiment A) and their conventional software development process (Experiment B).

Development times were recorded separately for all developers including all steps of DT development. The average times for each of these phases were calculated in Experiment A and Experiment B. Obtained results are presented in Figure 10 to facilitate comparing peer steps. Individual times can be found in (Dataset, 2019). The followings can be deduced for all steps of the development processes based on these results:

Analysis: The analysis step for each use case is not dependent on any tool or platform. Hence, the developers spent almost similar times in both Experiment A and Experiment B. Actually, this step is just dependent on the complexity of the embedded device's DT structure. Therefore, the difference between times can be ignored.

Design/Modelling: Elapsed time for modeling in Experiment A is slightly more than design in Experiment B. In addition to graphical modeling, DSML4DT has static semantics checks to avoid semantical errors which probably take more time to correct them later. Thus, modeling with DSML4DT extends the length of the design time slightly. This kind of detailed modelling was not considered by the developers during Experiment B. However, that additional time spent in modeling during Experiment A will lead to gain more detailed and accurate code generation in the next step.

Implementation/Generation: This is the step in which the developers achieved the most time saving with using DSML4DT. In Experiment A, DT code was obtained automatically for all use cases. In Experiment B, developers wrote required code manually based on their design. DSML4DT succeeded in the automatic generation of DT code by utilizing the products acquired from the detailed modeling. However, in Experiment A, the developers still needed to complete the generated code by manually adding delta code to have a fully executable program. Consequently, the implementation time required in Experiment B is about 5 times more than the generation and implementation time elapsed in Experiment A.

Test: Similar to the previous step, elapsed time in Experiment A for system tests was conspicuously lower than Experiment B. Developers tried to find syntactic and semantic errors of the DT programs. In Experiment B, developers needed to find errors in the entire code. In Experiment A, developers generally dealt only with delta code added, which is much less than the generated code. Because generated code constitutes most of the error-free and almost-ready final code which does not need to be re-validated. Hence, the time required to find errors during Experiment A is nearly 4 times less than Experiment B.

After completion of testing, all written DTs for all use cases were ready to be executed inside all targeted embedded systems. Hence, developers completed the creation of the DT components required for Driver Terminal, NVT, Validator and Multimedia Player systems. This holds for both experiments (A and B) although elapsed times for this step varied according to the production mechanism of DT components (with or without using DSML4DT) as discussed above.

Total Time: The total average development time elapsed in Experiment A is around 112 minutes when all case studies are considered. This time for Experiment B is around 235 minutes. Thus, developers completed the whole development process in Experiment A approximately 2 times faster than in Experiment B due to using DSML4DT.

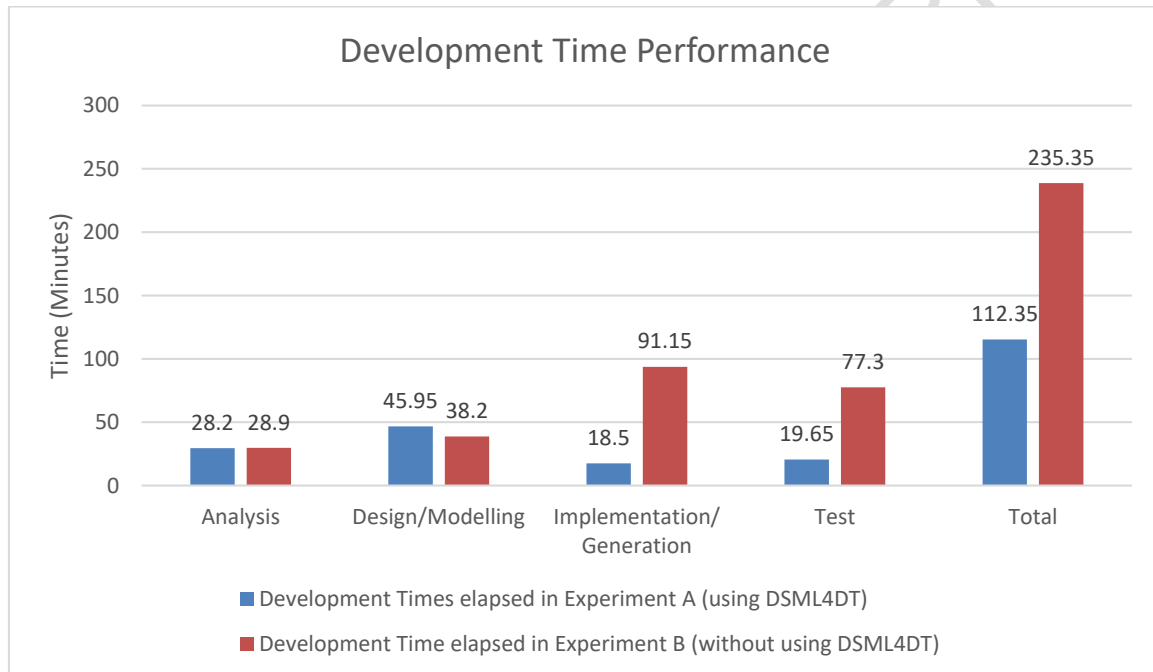


Figure 10: Comparison of the times elapsed during system development

6.1.2. Qualitative Assessment

Feedbacks, gained from the developers who experienced DSML4DT, were assessed to answer RQ3. For this assessment, a questionnaire was used which basically have 2 parts. In the first part, 26 questions were asked to the evaluators in Kentkart to assess their experience on using DSML4DT from different perspectives. To prepare this scoring part of the questionnaire, the Framework for Qualitative assessment of Domain-specific Languages (FQAD), introduced in

(Kahraman and Bilgen, 2015), was adopted and customized to the DSML4DT specifications. These questions were scored by the participants in the range of 1~5. Questions in this section are categorized into 10 different sections for evaluating a DSML: Functional Suitability, Usability, Reliability, Maintainability, Productivity, Extensibility, Compatibility, Expressiveness, Reusability and Integrability. In the second part of the questionnaire, 6 open ended questions were asked to the evaluators to criticize using DSML4DT. The whole questionnaire and all answers received from all evaluators are available at (Dataset, 2019).

Assessment according to Scored Questions:

The average scores received from the evaluators for the first part of the questionnaire are given in Figure 11. As can be seen, DSML4DT got scores ranging from 4 to 5 according to all assessment categories. The grand average of scores for all responses is 4.64 which is quite high. Hence, this result can be considered as one of the success indicators of the designed language.

Although average scores obtained for all categories are high, prominent features of DSML4DT for this assessment are its functional suitability, compatibility and reusability. Within this context, it seems that the evaluators agreed on DSML4DT's all-embracing model for DT structures and they found DSML4DT mostly suitable for the needs of various embedded system development. The scores also showed the evaluators thought DSML4DT promoting programming productivity. Moreover, DSML4DT was found compatible with DT specifications and hence DT models conforming to DSML4DT can be easily utilized during development. Finally, the evaluators confirmed DSML4DT's power on expressing DT structures and its language constructs can be re-usable for various DT applications.

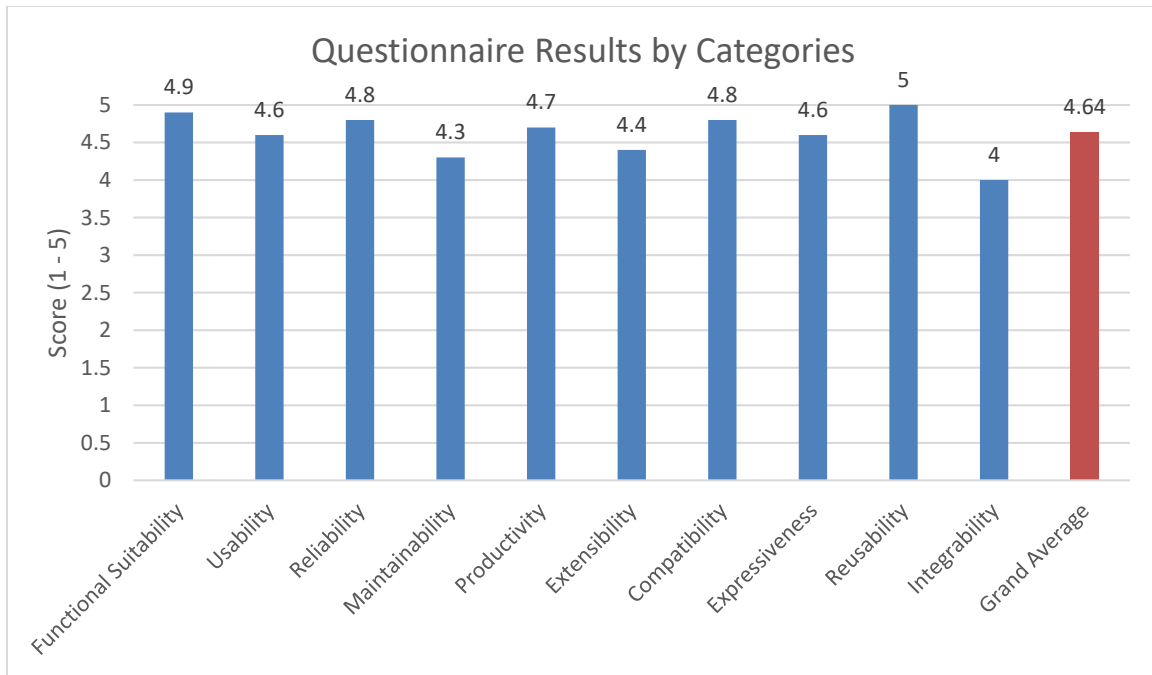


Figure 11: Scores according to assessment categories

Assessment according to Open-Ended Questions:

In the second part of the questionnaire, following questions were asked to the developers to get their feedbacks:

1. Does DSML4DT make DT software development easier?
2. Do you find DSML4DT useful for the development of DT-based embedded system software?
3. Do you think the DSML4DT structure is strong enough to model overall DT structure?
4. Do you think the DSML4DT IDE is easy to use?
5. Are there any difficulties you encountered while using DSML4DT? If so, do you have any suggestions to solve it?
6. Please write your suggestions and other comments for improving DSML4DT's features.

In answers for the first question, all evaluators agreed that DSML4DT simplifies DT software design and speeds up the development processes. For the second question, the evaluators found DSML4DT feasible while some of them specifically indicated visual modeling with DSML4DT facilitates embodying conceptual DT designs. Considering the third question,

majority of the evaluators stated built-in modeling brought by the IDE is very strong in support the sustainability of system development, i.e. DT models can be modified easily for changing hardware configurations. In all answers given to the fourth question, it seems the evaluators agreed that the IDE provides a simple and comprehensive design where users do not spend much effort on preparing DT programs.

In the fifth question, the problems encountered during the use of DSML4DT were asked. The evaluators stated that they do not encounter major problems. However, an evaluator said that installing the IDE for different computers is hard and time-consuming. We determined this difficulty originates from the configuration mismatches between the underlying Sirius platform and the setup of these computers. A detailed study has been carried out for this suggestion and a method of transporting DSML4DT IDE with underlying Eclipse Sirius platform as a software bundle has been derived. The installation of this new bundle now fixed this issue. Only two evaluators pointed out that migrating from classical software development methods into MDD came a little bit confusing at first and additional time is needed for adopting such approaches. Nevertheless, in their responses, the same evaluators also confirmed the features brought by DSML4DT and agreed on this new language facilitates the construction of DTs.

Some valuable suggestions were received for the sixth question. For example, an evaluator suggested to integrate modeling both DT structures and OS drivers. In fact, such an integration can be easily made inside DSML4DT's IDE when meta-entities pertaining to OS drivers are added into DSML4DT metamodel. For this purpose, a collaborative work with Kentkart engineers has been recently initiated on extending the metamodel considering Linux driver specifications. Another developer suggested highlighting the parts of the auto-generated DT programs where possible additions are needed. Considering this suggestion, DSML4DT's code generation process now produces an additional document informing which parts of the generated code need to be completed and marks the related code.

6.2. Threats to the validity

As with any evaluation study, there are some threats to the validity of the conducted evaluation. First, a relatively limited number of developers could participate in the assessment. However, DT software development in embedded systems is a specific field and the number of experts in this field is rare. Moreover, we paid attention on conducting this evaluation only with developers who actively implement commercial DT-based embedded

systems in the industry. As indicated at the beginning of this section, the developers participated in this study have significant experience on industrial scale DT development and we believe that both their experience and feedbacks contributed much on the evaluation of DSML4DT. Length and comprehensiveness of multi-case studies also affected the number of volunteers since the volunteer participants were requested to develop four different embedded software completely by following their conventional DT development approach first and then they had to repeat the development of each of these systems also with using DSML4DT. Nevertheless, the number of our participants is satisfying considering Nielsen's scale (Nielsen, 2012) for usability studies.

Second, single evaluator group was used instead of two different groups, which could pose a threat to the execution phase. In our previous studies (Yildirim and Kardas, 2014; Kardas et al., 2017; Kardas et al., 2018) for other domains, we experienced using both single and double evaluator groups. Using a single group may raise the risk that the DT development experience without using DSML4DT can be reflected into MDD with using DSML4DT (or vice versa) when the same system is to be developed twice. Using two groups may minimize this risk. However, in case of two groups, the qualitative evaluation based on the user feedbacks will not be completed in a fruitful way since the groups with or without using DSML4DT will be different. For the questionnaire-based comparison, it is crucial that a single group implements the same DT software with or without using DSML4DT. There is also the difficulty of creating two homogeneous groups which have almost same level of domain knowledge, experience and skills.

Finally, the choice of case studies may have an impact on the results. In order to mitigate this risk, instead of a single system development, multi-case studies with varying complexities were conducted. Rather than being trivial examples, multi-case studies herein consider actual embedded system DT implementations in a company operating in the relevant industry.

7. Conclusion

A DSML, called DSML4DT, for the MDD of DT software has been introduced in this paper. The difficulty of creating DT source files can be reduced and the necessity of mastering microprocessor-specific hardware while developing DTs can be eliminated by using

DSML4DT. After DT models are created, DT files can be generated automatically inside DSML4DT IDE.

A comparative evaluation of using DSML4DT was performed inside an IT company producing DT-based intelligent transportation systems. The results showed that approximately 76% of DT structures belonging to the hardware with different complexity can be obtained automatically only through modeling with DSML4DT. The new MDD process reduced the time elapsed for implementing software to half. Developers adopted the language particularly in terms of functional suitability, compatibility and reusability. The new DT software development process supported with DSML4DT is now being used in the company. DSML4DT language and IDE is available with including the required installation instructions at (Dataset, 2019).

In our future work, we aim at automatic integration of specific OS device drivers into DSML4DT models. Hence, custom driver definitions can be made available inside the graphical syntax of DSML4DT to the developers for specific devices and OS.

Acknowledgement

This work is funded by the Scientific and Technological Research Council of Turkey (TUBITAK) under grant 117E553. We would like to thank all staff in Kentkart Company for their valuable collaboration during the evaluations performed in this study and the directorate of Kentkart for their permission on conducting the evaluation studies.

References

AQL. 2018. “Acceleo Query Language”. <https://www.eclipse.org/acceleo/documentation/aql.html> (last access: November 2019)

Acceleo. 2018. “Acceleo Tool”. <https://www.eclipse.org/acceleo/> (last access: November 2019)

Arslan, S. and Kardas, G., Modeling Device Tree Software, In: Proc. 12th Turkish National Software Engineering Symposium, CEUR Workshop Proceedings, 2201, 2018, 1-12

Arslan, S., Turk, E. and Kardas, G., A Study on the Use of Device Tree Structures for Embedded Software Development, In: Proc. 2nd International Conference on Computer Science and Engineering, 2017, 882-887

Challenger, M., Kardas, G., Tekinerdogan, B. 2016. "A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems", *Software Quality Journal*, 24(3):755-795.

Chen, H., Godet-Bar, G., Rousseau, F., Petrot, F. 2014. "Device driver generation targeting multiple operating systems using a model-driven methodology", In proc. 25th IEEE International Symposium on Rapid System Prototyping, 30-36.

[dataset] (Dataset, 2019) Dataset for: DSML4DT: A domain-specific modeling language for device tree software, Mendeley Data, v1, 2019, <https://doi.org/10.17632/6d9nv4gk24.1> (last access: December 2019).

Devicetree Community. 2019. "The Devicetree Specification". <https://www.devicetree.org/>

Devigne, C., Brejon, J.-B., Meunier, Q., L., Wajsbürt, F. 2017. "Executing secured virtual machines within a manycore architecture", *Microprocessors and Microsystems*, 48:21-35.

Farhat, W., Sghaier, S., Faiedh, H., Souani, C. 2019. "Design of efficient embedded system for road sign recognition", *Journal of Ambient Intelligence and Humanized Computing*, 10(2):491-507.

Gioia, E., Passaro, P., Petracca, M. 2016. "AMBER: An advanced gateway solution to support heterogeneous IoT technologies", In proc. 24th International Conference on Software, Telecommunications and Computer Networks, 1-5

Jassi, M., Hu, Y., Mueller-Gritschneider, D., Schlichtmann, U. 2018. "Graph-Grammar-Based IP Integration (GRIP)—An EDA Tool for Software-Defined SoCs", *ACM Transactions on Design Automation of Electronic Systems*, 23(3), Article 40:1-26

Kahraman, G., Bilgen, S. 2015. "A framework for qualitative assessment of domain-specific languages", *Software & Systems Modeling*, 14(4):1505-1526.

Kardas, G., Bircan, E. and Challenger, M., Supporting the platform extensibility for the model-driven development of agent systems by the interoperability between domain-specific modeling languages of multi-agent systems, *Comput. Sci. Inf. Syst.* 14 (3), 2017, 875-912

- Kardas, G., Tezel, B. T. and Challenger, M., Domain-specific modelling language for belief-desire-intention software agents, *IET Software* 12 (4), 2018, 356-364.
- Katayama, T., Saisho, K., Fukuda, A. 2000. "Prototype of the device driver generation system for UNIX-like operating systems", In *proc. International Symposium on Principles of Software Evolution*, 302-310.
- Kelly, S., Tolvanen J.-P. 2008. *Domain-specific Modeling: Enabling Full Code Generation*. John Wiley & Sons
- Kentkart. 2019. *Kentkart Automatic Fare Collection & Vehicle Tracking Systems*, <http://www.kentkart.com/en> (last access: November 2019)
- King, M., Dave, N., Arvind. 2012. "Automatic generation of hardware/software interfaces", *ACM SIGPLAN Notices*, 47(4):325-336.
- Kosar, T., Mernik, M., Gray, J., Kos, T. 2014. "Debugging measurement systems using a domain-specific modeling language", *Computers in Industry*, 65(4):622-635
- Kosar, T., Bohra, S., Mernik, M. 2016. "Domain-Specific Languages: A Systematic Mapping Study". *Information and Software Technology*, 71:77-91
- Lecomte, S., Guillouard, S., Moy, M., Leray, P., Soulard, P. 2011. "A co-design methodology based on model driven architecture for real time embedded systems", *Mathematical and Computer Modelling*, 53(3-4):471-484
- Li, B., Bi, Y., He, Q., Ren, J., Li, Z. 2018. "A low-complexity method for authoring an interactive virtual maintenance training system of hydroelectric generating equipment", *Computers in Industry*, 100:159-172.
- Madiou, J. 2017. "The Concept of a Device Tree", 139-166. *Linux Device Drivers Development: Develop customized drivers for embedded Linux*. Packt Publishing
- Medini, K., Boucher, X. 2019. "Specifying a modelling language for PSS Engineering—A development method and an operational tool", *Computers in Industry*, 108:89-103.
- Nakamaru, T., Ichikawa, K., Yamazaki, T., Chiba, S. 2019. "Generating fluent embedded domain-specific languages with subchaining", *Journal of Computer Languages*, 50:70-83.

- Neuendorffer, S. 2018. "FPGA Platforms for Embedded Systems", In Nicolescu, G., Mosterman, P. J. (Eds.): Model-Based Design for Embedded Systems, CRC Press, 351-379.
- Nielsen, J. 2012. "How many test users in a usability study?". Nielsen Norman Group 4(6), <https://www.nngroup.com/articles/how-many-test-users/> (last access: November 2019)
- Nikkel, B. 2016. "NVM express drives and digital forensics", Digital Investigation, 16:38-45.
- Rocketboards. 2019. "Golden System Reference Design". <https://rocketboards.org/foswiki/view/Documentation/DeviceTreeGenerator> (last access: November 2019)
- Rodeh, O., Bacik, J., Mason, C. 2013. "BTRFS: The Linux B-Tree Filesystem", ACM Transactions on Storage, 9(3), Article 9:1-32
- Schüpbach, A., Baumann, A., Roscoe, T., Peter, S. 2012. "A Declarative Language Approach to Device Configuration", ACM Transactions on Computer Systems, 30(1), Article 5:1-35.
- Swaroop, K. N., Chandu, K., Gorrepotu, R., Deb, S. 2019. "A health monitoring system for vital signs using IoT", Internet of Things, 5:116-129
- The Eclipse Foundation. 2013. Eclipse Modeling Framework, <https://www.eclipse.org/modeling/emf/> (last access: November 2019)
- The Sirius Project. 2016. The Eclipse Sirius Modelling Project. <http://www.eclipse.org/sirius/> (last access: November 2019)
- Yildirim, O. and Kardas, G., A multi-agent system for minimizing energy costs in cement production, Comput. Ind. 65 (7), 2014, 1076-1084