*papageno*PCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping

Tobias Scheipel and Marcel Baunach

Institute of Technical Informatics
Graz University of Technology
Graz, Austria
E-mail: {tobias.scheipel, baunach}@tugraz.at

***Abstract*—Designing an embedded system from scratch is becoming increasingly challenging these days. In fact, the design process requires extensive manpower, comprising engineers with different fields of expertise. While most design principles for embedded systems start with a search for a suitable computing platform and the design of proper hardware to meet certain requirements, our novel vision involves using a completely different approach: The remainder of the embedded system can be automatically generated if the application software is available. In order to achieve automatic Printed Circuit Board (PCB) generation, we describe an approach, *papageno*PCB, in this paper, which is a part of a holistic approach called *papageno*X. *papageno*PCB provides a way to automatically generate schematics and layouts for printed circuit boards using an intermediate system description language. Therefore, the scope of the present work was to develop a concept, which could be used to analyze the embedded software and automatically generate the schematics and board layouts based on predefined hardware modules and connection interfaces. To be able to edit the plans once they have been generated, a file format for common electronic design automation applications, based on Extensible Markup Language (XML), was used to provide the final output.**

***Keywords*–embedded systems; printed circuit board; design automation; hardware/software codesign; systems engineering.**

I. Introduction

Embedded systems are relevant in almost every part of our society. From the simple electronics in dishwashers to the highly complex electronic control units in modern and autonomous cars – today, daily life is nearly inconceivable without those systems. As the technology improves, the complexity of embedded systems inevitably and steadily increases. A whole team of engineers usually plans, designs, and implements a novel system in several iteration steps. An example of such a process in the automotive industry is shown in Figure 1.

Designing an embedded system can be prone to errors due to a multitude of possible sources of such errors. This presents one major challenge when designing such a system: The challenge of how to eliminate error sources and make design processes more reliable and, therefore, cheaper. Nowadays, most design paradigms choose a bottom-up approach. This means that a suitable computing platform is chosen after defining all requirements with respect to these explicit requirements, prior experience, or educated guesses. Then, software development can either start based on an application kit of a computing platform, or some prototyping hardware must be built beforehand. If the requirements change during the development process, major problems could possibly arise,

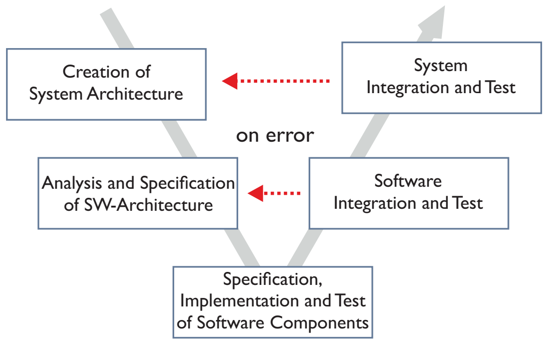
Creation of System Architecture
System Integration and Test
on error
Analysis and Specification of SW-Architecture
Software Integration and Test
Specification, Implementation and Test of Software Components

Figure 1. Automotive design process according to the V-Model [1].

e.g., new software features cannot be implemented due to computing power restrictions or additional devices cannot be interfaced because of hardware limitations. Another problem could arise if connection interfaces or buses become overloaded with too much communication traffic after the hardware has already been manufactured.

To tackle these problems, we propose a holistic approach, *papageno*X, and a sub-approach, *papageno*PCB, which are discussed in detail in the following sections. *papageno*X is a novel approach that has been developed for use while creating embedded systems with a top-down view. Therefore, it uses application source code to automatically generate the whole embedded system in hardware and software. One part of the concept behind this approach is *papageno*PCB. This concept handles the automatic generation of schematics and board layouts for printed circuit board design with standardized XML-based [2] output from intermediate system description models. To do so, a module-based description of the system hardware and software needs to be made. Furthermore, connections between the hardware modules on wire level are done automatically. The concept and its related challenges were the main topics of the work described in this paper, whereas software analysis and model generation is part of work that will be conducted in the future with *papageno*X.

The paper is organized as follows: Section II includes a summary of related work. The rough idea of the holistic vision of *papageno*X (this paper includes a detailed description of the first part of this concept) is illustrated in Section III, whereas Section IV starts with the system description format within *papageno*PCB. In Section V, an explanation is given of the necessary steps taken to create the final output, and Section VI includes a proof of concept example. The results of an analysis on the scalability and performance of the developed generator are presented in Section VII. The paper concludes with Section VIII, in which the steps that need to be taken to achieve a final version of *papageno*X are described.

II. Related Work

As this work dealt with the automatic generation of hardware and extensively utilized hardware definition models, it was influenced by existing solutions such as devicetree, which is used, e.g., within Linux [3]. The devicetree data structure is used by the target Operating System's (OS) kernel to handle hardware components. The handled components can comprise processors and memories, but also the internal or external buses and peripherals of the system. As the data structure is a description of the overall system, it must be created manually and cannot be generated in a modular way. It is mostly used with System-on-Chips (SoCs) and enables the usage of one compiled OS kernel with several hardware configurations. Different approaches have been taken to use annotated source code to extract information about the underlying system. Annotations can be used to analyze the worst-case execution times [4][5] of software in embedded systems. Other approaches that have been taken have used back-annotations to optimize the power consumption simulation [6]. These annotations have allowed researchers to gain a better idea of how the system works in a real-world application, meaning that the annotated information is based on estimations or measurements. As far as the automatic generation of schematics and board layouts is concerned, few solutions have been developed towards design automation. Some authors have dealt with the question of how to generate schematics using expert systems so their appearance is more pleasing to human readers [7]. Some work has even been carried out on the generation of circuit schematics by extracting connectivity data from net lists [8]. These approaches are all based on various kinds of network information and cannot be used to extract system data out of – or are even aware of – application source code or system descriptions.

All the approaches mentioned above have some advantages and inspired this work, as no solution has yet been proposed for how to automatically generate PCBs from source code.

III. Main Idea of *papageno*X

*papageno*X stands for **P**rototyping **AP**plication-based with **A**utomatic **GEN**eration **O**f **X**; it prospectively contains a toolchain that can be used to automatically generate the software, reconfigurable logic, and hardware of the final prototype of system X by simply using application software source code. In this context, system X could be an automotive Electronic Control Unit (ECU), a Cyber-Physical System (CPS), or an Internet-of-Things (IoT) device. After generating X, *papageno*X should also be able to check whether a new application version is still compatible with previously designed systems, or if it is not, which restrictions apply.

As depicted in Figure 2, *papageno*X uses Application Software (ASW) to generate software code that includes Basic Software (BSW) and an executable ASW, reconfigurable logic code in some hardware description language for Field Programmable Gate Arrays (FPGAs), as well as schematics

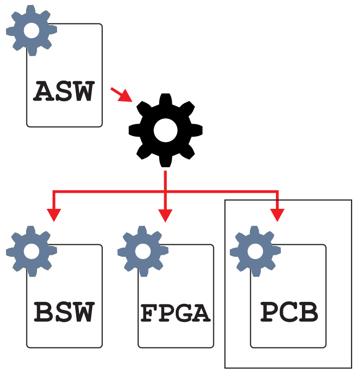

Figure 2. The main idea behind the *papageno*X approach.

and layouts for PCBs. In this context, the term BSW subsumes operating systems with, e.g., drivers, services. Even though the *papageno*X approach envisions generation of reconfigurable logic, it differs from, e.g., SystemC [9], because it also generates hardware on the PCB level.

In this paper, the very first step taken to generate a PCB from an intermediate system model (prospectively extracted from source code) is described.

IV. System Description Format

The system description format in *papageno*PCB is module-based. This means that every possible module, e.g., a Micro-controller Unit (MCU) board or different peripherals must be defined before they are connected with each other. The whole description and modeling approach taken is generic, which enables its easy adaptation to different use cases. The structure was defined according to a JavaScript Object Notation (JSON) [10] format, and three different kinds of definition files were established:

A. Module Definition: One single file that defines the hardware module, its interfaces and its pins, and a second file that contains the design block for creating schematics and board layouts concerning this module.

B. Interface Definition: Generic definition of several different interfaces between modules.

C. System Definition: Contains modules and connections between these; is abstractly wired with certain interface types.

All three types will be explained below. The example modules show footprints of a Texas Instruments (TI) LaunchPad^{TM} [11] with a 16-bit, ultra-low-power MSP430F5529 MCU [12].

A. Module Definitions and Design Blocks

The module definition of a TI LaunchPad^{TM} is shown in Figure 3. Apart from a name and a design block file property, this definition consists of an array of interfaces and pins. The design block file property refers to an EAGLE [13] design block file, comprising of a schematic placeholder (cf. Figure 4), and a board layout placeholder (cf. Figure 5). These placeholders will later be placed on the output schematics and board layouts. The array of interfaces may contain several different interface types of which the module is capable. The property *type* determines the corresponding interface type. In the case presented, two Serial Peripheral Interfaces (SPIs) are present. Both contain a name, the type *SPI*, and several pins.

```
1  {
2    name: "MSP430F5529_LaunchPad",
3    design: "MSP430F5529_LaunchPad.dbl",
4    interfaces: [{
5        name: "SPI0",
6        type: "SPI",
7        pins: { MISO: "P3.1", MOSI: "P3.0",
8          SCLK: "P3.2", CS: 'any@["P2.0", "P2.2"]'   }
9      }, {
10       name: "SPI1",
11       type: "SPI",
12       pins: { MISO: "P4.5", MOSI: "P4.4",
13         SCLK: "P4.0", CS: any }
14     }
15   ],
16   pins: ["P6.5", "P3.4", "P3.3", "P1.6",
17   "P6.6", "P3.2", "P2.7", "P4.2", "P4.1",
18   "P6.0", "P6.1", "P6.2", "P6.3", "P6.4",
19   "P7.0", "P3.6", "P3.5", "P2.5", "P2.4",
20   "P1.5", "P1.4", "P1.3", "P1.2", "P4.3",
21   "P4.0", "P3.7", "P8.2", "P2.0", "P2.2",
22   "P7.4", "RST" , "P3.0", "P3.1", "P2.6",
23   "P2.3", "P8.1"]
24 }
```

Figure 3. Module definition of a TI LaunchPad^{TM}with two SPI interfaces.

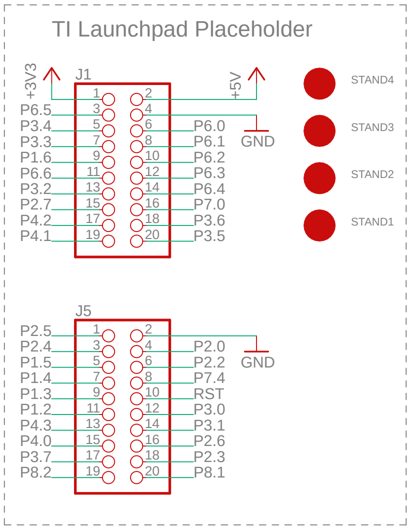

Figure 4. Schematics of a placeholder design block for a TI LaunchPad^{TM} [11].

Pins within interfaces can either be directly assigned to hardware pins (e.g., `MISO: "P3.1"` in line 7) or left for automatic assignment (e.g., `CS: any` in line 13). It is also possible to automatically assign a wire from a dedicated pool by using `any@somearray` (cf. line 8) syntax. Each module definition file is associated with its corresponding design block. It is of utmost importance that pin names are coherent in both module representations, as coherence of naming later ensures that proper interconnections are made between modules. Furthermore, a standard format for power supply connections must be used to avoid creating discrepancies between modules. The bus speed of the SPI was not taken into account in this work and will be addressed in future developments. As depicted in Figure 5, the board layout of a module only consists of its pins. The main idea here was to create a motherboard upon which modules can be placed using their exterior connections (e.g., pin headers or similar connectors). Therefore, the placeholder serves as interface layout between fully assembled PCB modules, such as the LaunchPad^{TM}, and can then be connected to other modules through interfaces.

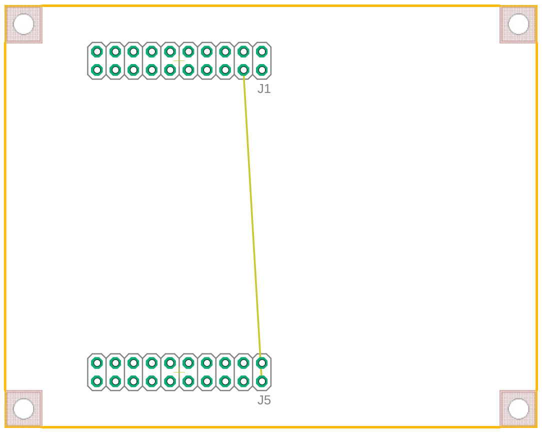

Figure 5. Board layout of a placeholder design block for a TI LaunchPad^{TM} [11].

```
1  {
2    interfaces: [
3      {
4        type: "SPI",
5        connections: [
6          { "master.MOSI" : "bus.MOSI" },
7          { "master.MISO" : "bus.MISO" },
8          { "master.SCLK" : "bus.SCLK" },
9          { "slave.MOSI"  : "bus.MOSI" },
10         { "slave.MISO"  : "bus.MISO" },
11         { "slave.SCLK"  : "bus.SCLK" },
12         { "master.CS"   : "wiremultiple" },
13         { "slave.CS"    : "wiresingle" }
14       ]
15     }
16   ]
17 }
```

Figure 6. Interface definition containing SPI.

B. Interface Definitions

After defining the modules, the generic interfaces must be defined. The interface definition collection is centralized in a single file, and its structure is shown in Figure 6. In this example, only SPI [14] has been defined with its standard connections. As the format is generic, other interface types, e.g., Inter-Integrated Circuit (I^2C, [15]) or even a Controller Area Network (CAN, [16]), are also feasible. It also shows how masters and slaves within this communication protocol are connected to the bus wires. As the SPI also has so-called Chip Select (CS) wires for every slave selection, special treatment must be used here: A slave only has one CS wire, which is marked with *wiresingle* (cf. line 13), whereas a master has as many CS wires as it has slaves connected to it (marked with *wiremultiple*; cf. line 12).

C. System Definition

The final step taken was to define the system itself, which was built from modules and the connections between them. To do so, a single project file must be created, as illustrated in Figure 7. Initially, all necessary modules are imported and named accordingly within the *modules* array. Once defined, they can be interconnected using the previously defined interface definitions. In our example, LaunchPad^{TM} *MSP1* was connected to a microSD board *SD1* of type ”MicroSD Breakout Board” [17] via SPI. This particular SPI connection is called *SPI_Connection1* of type *SPI* and has two participants with different roles: *MSP1* as a master and *SD1* as a slave. This system definition will prospectively be generated and extracted

```
{
  modules: [
    { name: "MSP1",
      type: "MSP430F5529_LaunchPad"},
    { name: "SD1",
      type: "MicroSD_BreakoutBoard"}
  ],
  connections: [
    {
      name: "SPI_Connection1",
      type: "SPI",
      participants: [
        { name: "MSP1", role: "master" },
        { name: "SD1", role: "slave" }
      ]
    }
  ]
}
```

Figure 7. A system model containing two modules connected via SPI.

out of the ASW code by *papageno*X. The *papageno*PCB approach is taken to generate PCBs only.

V. Implementation of PCB Generation

After having defined the modules, interfaces, and implemented a system definition, PCB generation can start. The generation consists of two major steps: (*A.*) establishing connection wires based on predefined module and system definitions, and assigning dedicated pins and (*B.*) generating XML-based schematic files from its output. The final step (*C.*), which is carried out to deal with the final layout of the schematics, must be done (in part manually) afterwards. The generator is developed as a Java command line application to maintain platform-independence and ensure that it can be integrated into standard tool chains and build management tools.

A. Connection Establishment and Pin Assignment

During this first step, JSON data structure analysis presents the main challenge. The whole system must be interconnected appropriately using the previously explained definition files. To do so, all connections within the system definition must be matched at the beginning of the process. This task subsumes the discovery of connections between modules, their mapping to certain interface types, and the final wire allocation required to interconnect all participants. Specifically, each connection has a type and a finite number of participants with different roles, interfaces, and pins. These pins must then be connected to the newly introduced wires, belonging to the communication. Several different types of wires can be used to connect the participants with each other:

The easiest wires to use are common wires, which can be assigned to a pool of free pins of the module. These wires are marked with *wiresingle* within the interface definition. Due to the fact that all unused General-Purpose Input/Output (GPIO) pins of a module can be used for this purpose, they need to be assigned last.

Furthermore, every participant can connect itself directly to bus wires via its dedicated pins, depending on, e.g., the type of MCU used. In the case of an MSP430 MCU, certain pins are electrically connected to an interface circuit, as defined in its module definition (cf. Figure 3). These pins must, therefore, be matched with the connection's wires (cf. Figure 6). The interface definition must match roles and pins accordingly to correctly interconnect the participants of each connection.

Another type of wires that can be used are multiple wires. If we take SPI as an example, the master needs to have as many chip-select wires as slaves with which it wants to communicate. Therefore, this type of wire – marked with *wiremultiple*, as previously defined – must clone itself to obtain the number of wires needed.

These different types of wires must be connected to the pins of the modules to establish a proper connection or *net* according to the interface definition. The interconnected modules with their nets form a holistic JSON-based description of the system.

B. Schematic and Board Layout Generation

Utilizing the interconnected system description, schematics and board layouts can be generated. In our case, EAGLE's XML data structure [2] was used to form a dedicated output file for schematics and board layouts. To generate those plans, (1) design blocks for each module must be loaded, (2) the previously found connections must be applied and (3) the connected design blocks must be placed on an empty schematic plan or board layout.

(1) In this step, each module has to be instantiated by loading the corresponding design block of its type.
(2) This step must carried out to form the whole system according to the JSON-based holistic description. Therefore, pins of each module must be assigned to the wires of a connection within the system. To do so, each connection again must be applied separately to each participant. As the system description already contains information, as to which pin of a module must be connected to which wire, this can be done quite easily.
(3) This step, which is the computationally most expensive step, must be carried out to merge the connected instances of each module into an empty plan, as a great deal of XML parsing is required here. To create consistent plans, the design blocks must be prepared well beforehand to avoid, e.g., inconsistencies within board layers or signal names. To keep the modules from overlapping, a two-dimensional translation of each module must be executed as part of each merge procedure as well. In total, two merging steps are required for each module – one for the schematic and one for the board layout. As this approach generates connection PCBs (”motherboards”) where one can plug in modules, only placeholders are used.

Finally, the two generated XML structures are exported and saved into different files for further usage.

C. Routing Generated Schematics and Board Layouts

As layouting and routing of PCBs is a non-trivial task, and engineers need a great deal of experience when performing a task like this, *papageno*PCB cannot be used to produce final variants of a board. It is recommended to use EAGLE's auto-routing functionality or manual routing to finalize the already well-prepared layouts.

VI. Proof of Concept

The proof of concept comprises the generation of the system definition as shown in Figure 7. As mentioned before, the system created consists of two modules interconnected with

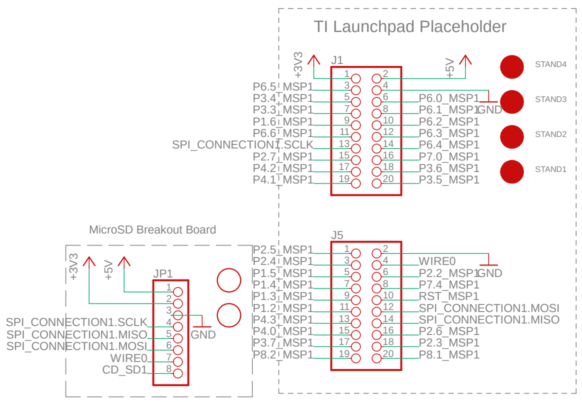

Figure 8. Raw output of the schematics generated as displayed in EAGLE.

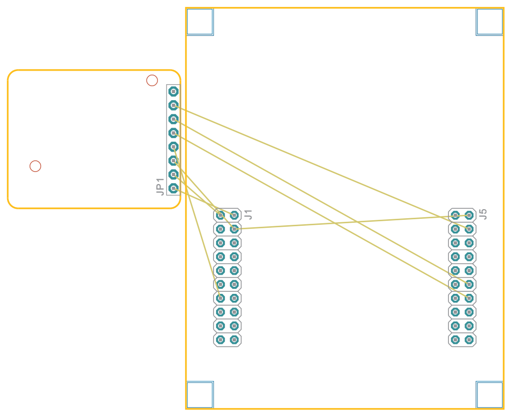

Figure 9. Raw output of the board layout generated as displayed in EAGLE.

one SPI bus, whereas the processor board serves as master. The schematics generation step yields in the drawing depicted in Figure 8.

Compared with the LaunchPad^{TM}'s design block shown in Figure 4, one can see the differences in the net names. As examples, *P2.0* has been replaced with *WIRE0*, and *P3.0* is now assigned to *SPI_CONNECTION1.MOSI*. These wires connect to pins 7 and 6 of the MicroSD Breakout Board on the left, respectively. Also, each unconnected pin gets a suffix describing its module (cf. *_MSP1*). These newly introduced net names are the results of the wire generation explained in Section V-A. As the reusability of schematic plans is an important aspect, the feature of non-overlapping module placement can be emphasized as well. The result of the board layout generation step is shown in Figure 9, as described in Section V-B. The fine lines show non-routed connections between the pins. As the plan will be manufactured as a real hardware PCB, no part can overlap. Routing of the board has to be either performed manually or by using a design tool's built-in auto router. A feasible layout variant is presented in Figure 10. EAGLE can also be used to check the correctness of the XML file format.

VII. Scalability and Performance

In this Section, we describe measurements and investigations that concern the performance of the PCB-generating process. All discussed evaluations use one setup as a reference. The application was executed with a Java 10 virtual machine on an Intel Core i7 7500U@2.7GHz with 16 gigabytes of RAM. Table I shows the mean execution time and the

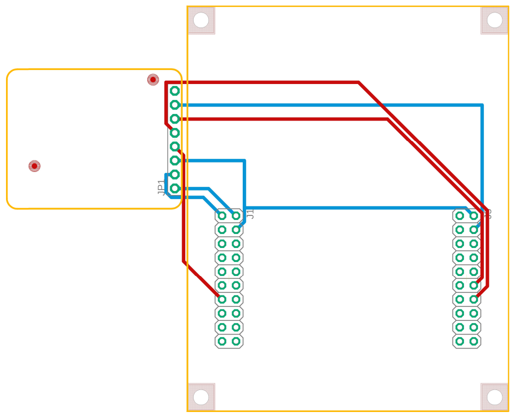

Figure 10. Board layout after auto-routing in EAGLE.

combined output XML file size of the generation process of different test case scenarios, which are explained below. All test cases featured a different number of participants (part.) which consisted of masters (M) and slaves (S) with different connection types.

TABLE I. MEAN EXECUTION TIMES FOR DIFFERENT SCENARIOS.

#	*test scenario description*	*execution time*	*file size*
	1 SPI conn. (scenario 1)		
0	2 part. (1 M, 1 S)	678.98 ms	94 KiB
1	3 part. (1 M, 2 S)	793.56 ms	100 KiB
2	4 part. (1 M, 3 S)	893.89 ms	105 KiB
3	5 part. (1 M, 4 S)	983.56 ms	110 KiB
4	6 part. (1 M, 5 S)	1 060.70 ms	116 KiB
5	7 part. (1 M, 6 S)	1 151.37 ms	122 KiB
	2 SPI conn. (scenario 2)		
0	4 part. (1 M and 1 S each)	910.93 ms	115 KiB
1	6 part. (1 M and 2 S each)	1 079.78 ms	126 KiB
2	8 part. (1 M and 3 S each)	1 242.88 ms	137 KiB
3	10 part. (1 M and 4 S each)	1 388.39 ms	149 KiB
4	12 part. (1 M and 5 S each)	1 500.44 ms	160 KiB
5	14 part. (1 M and 6 S each)	1 613.93 ms	171 KiB
	1 I^{2}C conn. (scenario 3)		
0	2 part. (1 M, 1 S)	693.04 ms	105 KiB
1	3 part. (1 M, 2 S)	813.64 ms	111 KiB
2	4 part. (1 M, 3 S)	918.96 ms	117 KiB
3	5 part. (1 M, 4 S)	1 010.40 ms	123 KiB
4	6 part. (1 M, 5 S)	1 107.56 ms	129 KiB
5	7 part. (1 M, 6 S)	1 198.17 ms	135 KiB
	1 I^{2}C and 1 SPI conn. (scenario 4)		
0	3 part. (1 M, 1 S each)	828.13 ms	127 KiB
1	5 part. (1 M, 2 S each)	1 020.54 ms	138 KiB
2	7 part. (1 M, 3 S each)	1 195.82 ms	150 KiB
3	9 part. (1 M, 4 S each)	1 337.50 ms	162 KiB
4	11 part. (1 M, 5 S each)	1 493.75 ms	173 KiB
5	13 part. (1 M, 6 S each)	1 612.04 ms	185 KiB

Each test case is based on the example described in Section VI but with different constellations concerning the numbers and types of participants and connections. All test cases were executed 100 times. Four types of test scenarios with six test cases each were conducted: Within the first scenario, just one SPI connection was present, with a varying number of slaves each test case. The second scenario comprised two SPI connections with an increasing number of slaves. Test scenario three had one I^{2}C connection and was similar to scenario one, whereas scenario four included SPI and I^{2}C connections to a single master with an increasing number of slaves. The devolution of the mean execution time (in ms) in all test scenarios is shown in Figure 11. When comparing all scenarios, the trend observed is quite similar: All performance graphs show a linear devolution with an additive, logarithmic-

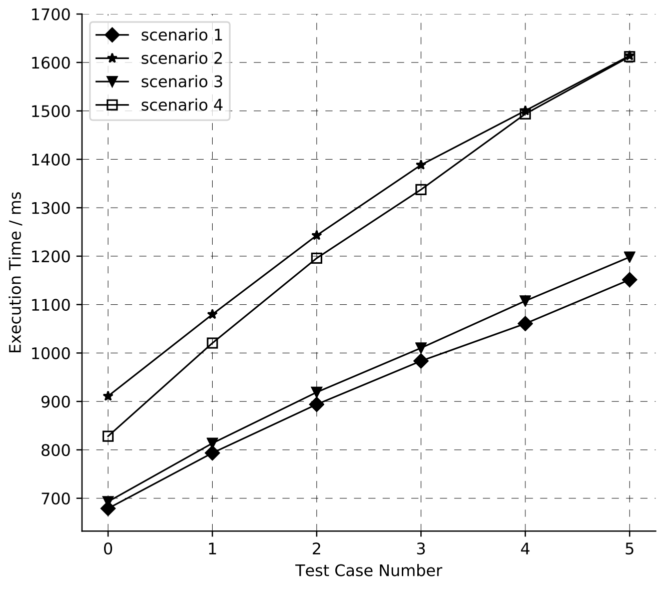

Figure 11. Performance graph for different test cases.

like component. The linear component is due to the linear increase in the complexity of the test cases. The logarithmic-like growth observed can be explained by the decreasing, additive overhead of the linear component when processing similar connection reasoning, as well as the XML schematic and layout data. This is also the reason why doubling the numbers in the first test case yielded in much higher values than in test case two. Test scenario four is the only one that displays a steeper curve. This is due to the combination of different connection types, yielding less-optimal algorithm executions. As XML processing is quite costly, some further optimizations are needed. As the overall file size displayed linear growth, no correlation was observed between file size and execution time.

VIII. Conclusion and Future Work

In conclusion, *papageno*PCB represents a novel, top-down approach that can be taken to develop an embedded system. With just having a model-based system description at hand, it is possible to use *papageno*PCB to generate hardware schematics and board layouts accordingly. This opens up a numerous new possibilities on higher abstraction levels. It is feasible to carry out automatic bus balancing or bandwidth engineering before building the hardware. The use of these concepts requires the availability of in-depth information about the electrical and mechanical characteristics of all parts of a PCB, so that the hardware can be optimized regarding non-functional metrics such as bandwidth or power consumption. Due to the generic design, new models can be integrated easily, and it will be even possible to take a non-module-based approach on the device level (if the proper definitions are available).

In the future, work must be carried out to extract system models from ASW source code. Therefore, we are working on introducing annotations into our operating system environment [18], which will enable us to automatically generate system definition files. These annotations can either be introduced into the code as compiler keywords (e.g., pragmas, defines) or as comments. As some work is already being conducted to improve the automatic portability of real-time operating systems [19], the proposed approach could be used to build a system for which only the application code must be programmed. The rest of the system can then be generated automatically. Even suitable and application-optimized processor architectures [20] could be created by taking this approach. The ultimate goal is to establish *papageno*X as a universal embedded systems generator using only annotated ASW code.

Acknowledgment

This research project was partially funded by AVL List GmbH, the Austrian Federal Ministry of Education, Science and Research (bmbwf), and Graz University of Technology.

References

[1] J. Schäuffele and T. Zurawka, *Automotive Software Engineering*, ser. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2016.

[2] Autodesk, Inc., *EAGLE XML Data Structure 9.1.0*, 2018.

[3] devicetree.org, *Devicetree Specification*, Dec. 2017, release v0.2.

[4] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance Timing Simulation of Embedded Software," in *Proc. of the 45th Annual Design Automation Conference*, June 2008, pp. 290–295.

[5] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener, "Embedded Program Annotations for WCET Analysis," in *WCET 2018: 18th Int'l Workshop on Worst-Case Execution Time Analysis*, Barcelona, Spain, Jul. 2018, pp. 8:1–8:13.

[6] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, Retargetable Back-annotation for Host Compiled Performance and Power Modeling," in *Int'l Conference on Hardware/Software Codesign and System Synthesis*, Piscataway, NJ, USA, 2013, pp. 1–10.

[7] G. M. Swinkels and L. Hafer, "Schematic generation with an expert system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 12, pp. 1289–1306, Dec 1990.

[8] B. Singh *et al.*, "System and method for circuit schematic generation," US Patent US7 917 877B2, 2011.

[9] IEEE Standards Association, *IEEE 1666-2011 - IEEE Standard for Standard SystemC Language*, Sep. 2012.

[10] ECMA International, *ECMA-404: The JSON Data Interchange Syntax*, 2nd ed., Dec. 2017.

[11] Texas Instruments, *MSP430F5529 LaunchPad^{TM} Development Kit (MSP--EXP430F5529LP)*, Apr. 2017.

[12] ——, *MSP430x5xx and MSP430x6xx Family User's Guide*, Mar. 2018, [retrieved: Jan, 2019]. [Online]. Available: http://www.ti.com/lit/ug/slau208q/slau208q.pdf

[13] Autodesk, Inc., "EAGLE," [retrieved: Jan, 2019]. [Online]. Available: https://www.autodesk.com/products/eagle/

[14] S. Hill *et al.*, "Queued serial peripheral interface for use in a data processing system," US Patent US4 816 996, 1989.

[15] NXP Semiconductors, Inc., *UM10204: I2C-bus specification and user manual*, Apr. 2014, rev. 6.

[16] International Organization for Standardization, *ISO 11898: Road vehicles – Controller area network (CAN)* , 2nd ed., Dec. 2015.

[17] Adafruit Industries, *Micro SD Card Breakout Board Tutorial*, Jan. 2019, [retrieved: Jan, 2019]. [Online]. Available: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-micro-sd-breakout-board-card-tutorial.pdf

[18] R. Martins Gomes, M. Baunach, M. Malenko, L. Batista Ribeiro, and F. Mauroner, "A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems," in *Proc. of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017, pp. 41–46.

[19] R. Martins Gomes and M. Baunach, "A Model-Based Concept for RTOS Portability," in *Proc. of the 15th Int'l Conference on Computer Systems and Applications*, Oct. 2018, pp. 1–6.

[20] F. Mauroner and M. Baunach, "mosartMCU: Multi-Core Operating-System-Aware Real-Time Microcontroller," in *Proc. of the 7th Mediterranean Conference on Embedded Computing*, Jun. 2018, pp. 1–4.