Towards an Automated Printed Circuit Board Generation Concept for Embedded Systems

Tobias Scheipel and Marcel Baunach

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

***Abstract*—Future embedded systems will need to be generic, reusable and automatically adaptable for the rapid advance development of a multitude of different scenarios. Such systems must be versatile regarding the interfacing of electronic components, sensors, actuators, and communication networks. Both the software and the hardware might undergo a certain evolution during the development process of each system, and will significantly change between projects and use cases. Requirements on future embedded systems thus demand revolutionary changes in the development process. Today these processes start with the hardware development (bottom-up). In the future, it shall be possible to only develop application software and generate all lower layers of the system automatically (top-down). To enable automatic Printed Circuit Board (PCB) generation, the present work deals mainly with the question *"How to automatically generate the hardware platform of an embedded system from its application software?"*. To tackle this question, we propose an approach termed *papageno*PCB, which is a part of a holistic approach known as *papageno*X. This approach provides a way to automatically generate schematics and layouts for printed circuit boards using an intermediate system description format. Hence, a system description shall form the output of application software analysis and can be used to automatically generate the schematics and board layouts based on predefined hardware modules and connection interfaces. To be able to edit and reuse the plans after the generation process, a file format for common electronic design automation applications, based on Extensible Markup Language (XML), was used to provide the final output files.**

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

I. INTRODUCTION

Embedded systems are of relevance in virtually every area of our society. From the simple electronics in dishwashers to the highly complex electronic control units in modern and autonomous cars – daily life today 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 error sources. 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. Most design paradigms today 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, 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 already proposed a holistic approach, ***papageno***X, and a sub-approach, ***papageno***PCB in [1]. In the course of this work, we intend to further discuss and extend our approach in more 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 PCB 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, an analysis of the scalability and performance for the developed generator and a use case with a manufactured prototype. The paper concludes with Section VII, in which the steps that need to be taken to achieve a final version of ***papageno***X are described.

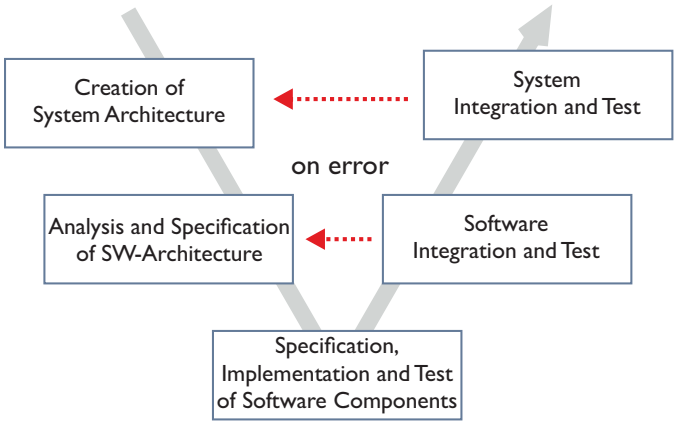

Figure 1. Automotive design process according to the V-Model [3].

II. State of the Art and Related Work

This section gives an overview on how embedded systems are developed nowadays and how hardware can be generated automatically in different types of systems. Additionally, some approaches towards software annotations and design space exploration are shown to provide an overview.

A. Embedded Systems Prototyping Approaches

Conventional embedded systems prototyping makes use of very specialized hardware platforms, capable of executing a vast variety of use cases typical for the field of deployment (e.g., an automotive Electronic Control Unit (ECU), a Cyber-Physical System (CPS), an Internet-of-Things (IoT) device).

In the context of ECU prototyping platforms, one approach is *rCube2* [4], based on two powerful independent TC1797 [5] Microcontroller Units (MCUs). The two processors can interact via shared memory, but are completely isolated during execution. AVL RPEMS [6] is a generic engine control platform provided as a highly flexible and configurable engine management system for the development and optimization of conventional and new combustion engines, power and emissions optimization, and the realization of hybrid and electric powertrains. The current version consists of a single-core automotive MCU (TC1796 [7] or TC1798 [8]) with different variants for diesel and gasoline engine control applications. These different PCBs are equipped with automotive-compliant Application-specific Integrated Circuits (ASICs) and a head-mounted MCU board, which allows prototyping as close as possible to series production. It was designed to offer engineers utmost flexibility when developing new control algorithms for non-standard engines, or standard engines with new components.

There are other similar prototyping platforms from commercial suppliers, but they also lack in flexibility and adaptability when it comes to hardware changes. The main problem of those commercial solutions is that even though they offer high performance and come with complete toolchains, their hardware is very different from a series device, as overcompensation takes place. Since the components cannot be easily changed when a prototype is turned into a commercial product, a complete redesign has to take place.

When the need for hardware changes after deployment is taken into account, reconfigurable logic is mostly mentioned in literature. This can reach from pure Field Programmable Gate Arrays (FPGAs) to System on Chips (SoCs), which include an FPGA alongside other MCU cores (e.g., Zynq-7000 [9]). The main advantage of those systems is that one does not have to change the physical hardware, but can easily adapt features like on-chip peripherals within the logic without the need of manufacturing an ASIC. However, it is not possible to change physical hardware features after deployment with those devices.

B. Automatic Hardware Generation

As this work is concerned 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 [10]. 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 SoCs and enables the usage of one compiled OS kernel with several hardware configurations. As far as automatic generation of schematics from software is concerned (top-down), there are a few solutions towards design automation. Some papers deal with the question, how to generate schematics, so that these look nice for a human reader by using expert systems [11]. Some work on the generation of circuit schematics has even been done by extracting connectivity data from net lists [12]. These approaches all have some kind of network information as a basis and do not extract system data out of – or are even aware of – application source code or system descriptions.

C. Annotations and Design Space Exploration

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 [13][14] of software in embedded systems. Other approaches that have been taken have used back-annotations to optimize the power consumption simulation [15]. 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. Introduction of annotations can be achieved by simple source code analysis or more sophisticated approaches such as, e.g., creating add-ons or introducing new features into source code compilers.

To generate systems out of application software, annotations can be used to extract requirements. These requirements can then be utilized to apply design space exploration [16] by, e.g., modeling constraints [17]. In [18], different types for design space explorations are shown and categorized, also mentioning language-based constraint solvers featuring, e.g., MiniZinc [19]. By using approaches like these, a design space model can easily be translated into a mathematical model for optimization.

All the approaches and concepts 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

The main idea of ***papageno***X consists of an application driven electronics generation and the inversion of the state of the art "software follows or adjusts to hardware" paradigm in embedded systems development, where the design starts with the hardware architecture. Software is then built on selected components (e.g., automotive grade MCUs and PCBs).

Even when hardware deficits become visible during the software development process, the hardware is unlikely to see significant changes due to the high cost and many people or even companies involved. Thus, software developers try to compensate, e.g., by manual tuning and workarounds beyond automotive standards (e.g., AUTOSAR [20]). This violates compliance and is one reason why prototypes differ significantly from series devices, also complicating the transition and the subsequent maintenance in the field. Apart from this, future embedded systems will contain reconfigurable logic which is scarcely supported in current development processes due to both the lack of support and a fear of even more complexity (in addition to the software, electronics, and networks). This is why ***papageno***X is an abbreviation for **P**rototyping **AP**plication-based with **A**utomatic **GEN**eration **O**f **X**. The envisioned concept of it will prospectively contain a set of tools 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 ECU, a CPS, an IoT device or some other embedded system. The goal is to support frequent changes to the Application Software (ASW) requirements by immediately reflecting them in the Basic Software (BSW), logic, and electronics – reducing time to market and efforts in development and maintenance. During development, the process will optimize the selection and configuration of BSW, on-board components, network interfaces, etc. for simplified transition to series production ("perfect fit"). After deployment, the process will help in the assessment of intended ASW chances to quantify the consequences on lower layers and thus to evaluate their feasibility and cost.

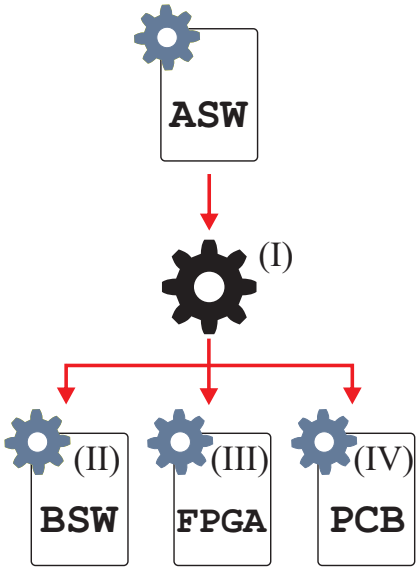

Figure 2. The main idea behind the ***papageno***X approach.

As depicted in Figure 2, the starting point of ***papageno***X is some application as a model or in source code. This ASW is analyzed in order to get to know all necessary requirements for the underlying system layers. These requirements are then used to generate software code that includes BSW and an executable ASW, reconfigurable logic code in a hardware description language (e.g., VHDL [21], Verilog [22]) for FPGAs, as well as schematics and layouts for PCBs. In this context, the term BSW subsumes operating systems with, e.g., drivers, services, hardware abstraction. Even though the ***papageno***X approach envisions the generation of reconfigurable logic, it differs from, e.g., SystemC [23], because it also generates hardware on the PCB level.

The following steps are envisioned within ***papageno***X:

1) application software development
2) in-depth analysis of ASW with respect to functional and non-functional requirements (NFRs)
3) creation of a selection space over potential components
4) filtering of the selection space with respect to general design decisions (e.g., data retention time)
5) generation of potential configurations from components
6) evaluation and optimization towards NFRs to select a single or several final, best fitting configuration(s)
7) mapping of functions or algorithms to reconfigurable logic (FPGAs)

To get a simple overview, the following example sketches the envisioned process while developing an embedded system with our novel approach: a user wants to store data somewhere permanently; with a data rate $\geq 5MB/s$ by writing this line of code in the ASW:

```
store_data(&data, StoreType.Permanent, 5000000);
```

The follow-up analysis of the ASW yields in an exemplary selection space as depicted in Figure 3. The green filled boxes illustrate the final configuration selected by the concept.

So, apart from the running application on the topmost level, a BSW must be generated, supporting a FAT16 [24] file system on top of a SD card driver and its underlying Serial Peripheral Interface (SPI, [25]) module driver. But even more important for this work, the final embedded system must be composed of a computing platform and a storage device, interconnected with each other. Finally, the generated system structure must be manufactured on a PCB, still matching all requirements with its properties.

Based on this overview, ***papageno***X will contain four major parts (also depicted in Figure 2):

(I) ASW analysis → creates selection space
(II) BSW generation → derives, e.g., needed components, drivers, OS features
(III) FPGA generation → maps functions to reconfigurable logic
(IV) PCB generation → module-based generation of suitable PCBs

In this paper, however, the main focus is on (IV), where a very first step is taken to generate a PCB from an intermediate system model (prospectively extracted from source code). The attempt is made to answer the research questions "What information is needed to automatically generate PCBs from ASW?" and "How can this information be used to generate a PCB prototype matching all ASW requirements?". It is henceforth named ***papageno***PCB.

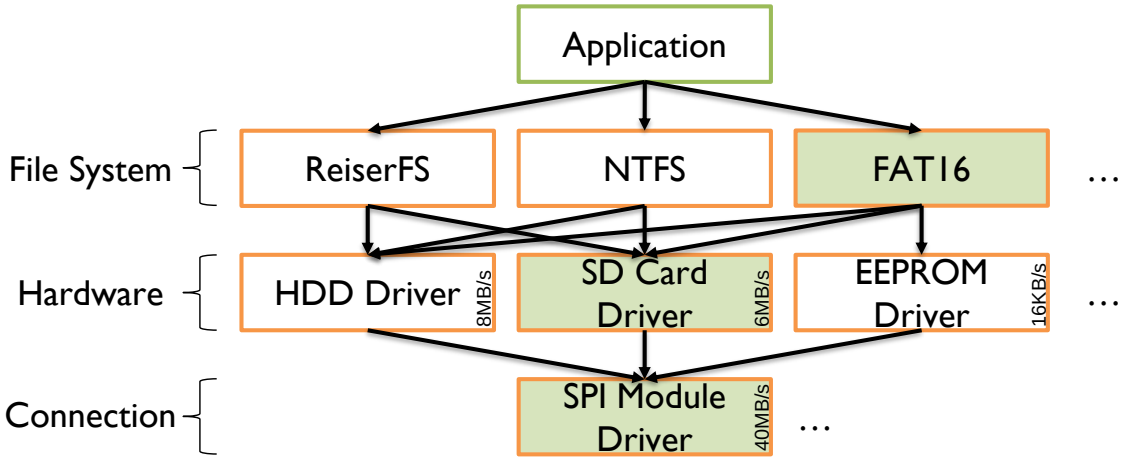

Figure 3. A system's requirements mapped to a corresponding exemplary selection space.

IV. System Description Format

The system description format in ***papageno***PCB is module-based. This means that every possible module, e.g., a 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) [26] 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 interface types to interconnect modules with each other; new types can be easily implemented and included within this file.

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 (1) a Texas Instruments (TI) LaunchPad™ [27] with a 16-bit, ultra-low-power MSP430F5529 MCU [28] and (2) MicroSD card module of type "MicroSD Breakout Board" [29].

```
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       name: "I2C0",
16       type: "I2C",
17       pins: { SDA: "P3.0", SCL: "P3.1" }
18     }, {
19       name: "I2C1",
20       type: "I2C",
21       pins: { SDA: "P4.1", SCL: "P4.2" }
22     }
23   ],
24   pins: ["P6.5", "P3.4", "P3.3", "P1.6",
25   "P6.6", "P3.2", "P2.7", "P4.2", "P4.1",
26   "P6.0", "P6.1", "P6.2", "P6.3", "P6.4",
27   "P7.0", "P3.6", "P3.5", "P2.5", "P2.4",
28   "P1.5", "P1.4", "P1.3", "P1.2", "P4.3",
29   "P4.0", "P3.7", "P8.2", "P2.0", "P2.2",
30   "P7.4", "RST" , "P3.0", "P3.1", "P2.6",
31   "P2.3", "P8.1"]
32 }
```

Figure 4. Module definition of a TI LaunchPad™with two SPI and two I²C interfaces, both overlapping.

A. Module Definitions and Design Blocks

The module definition of (1) a TI LaunchPad™ is shown in Figure 4, whereas the definition of (2) a MicroSD Breakout Board can be seen in Figure 5. 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 [30] design block file, comprised of a schematic placeholder (cf. Figures 6 and 7, respectively), and a board layout placeholder (cf. Figures 8 and 9, respectively). 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 module (1) in Figure 4, two SPIs and two Inter-Integrated Circuit (I²C, [31]) interfaces are present. Both contain a name, the type (*SPI*, *I2C*), and several pins. Module (2) in Figure 5, on the contrary, is very simple, with only one SPI interface in total.

Pins within interfaces can either be directly assigned to hardware pins (e.g., `MISO: "P3.1"` in line 7, Figure 4) or left for automatic assignment (e.g., `CS: any` in line 13, Figure 4). It is also possible to automatically assign a wire from a dedicated pool by using `any@somearray` (cf. line 8, Figure 4) syntax. This syntax enables the placing of so-called Chip Select (CS) wires in a more detailed way, e.g., based on needs for shorter connection wires, module specifications or other PCB properties. In this case, `somearray` must, of course, be replaced by a JSON-compliant array of strings, being a subset of the pins of the module, cf. Equation (1).

$$somearray \subseteq pins \tag{1}$$

```
1  {
2    name: "MicroSD_BreakoutBoard",
3    design: "MicroSD_BreakoutBoard.dbl",
4    interfaces: [
5      {
6        name: "SPI1",
7        type: "SPI",
8        pins: { MISO: "DO", MOSI: "DI",
9          SCLK: "CLK", CS: "CS" }
10     }
11   ],
12   pins: ["CLK", "DO", "DI", "CS", "CD"]
13 }
```

Figure 5. Module definition of a MicroSD Breakout Board with an SPI interface.

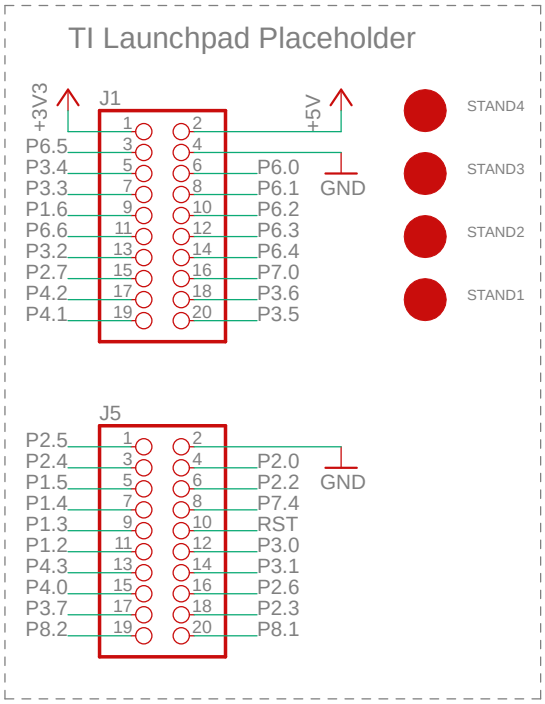

Figure 6. Schematics of a placeholder design block for a TI LaunchPad™ [27].

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 the naming coherence 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 and the I²C was not considered in this work and will be addressed in future developments towards NFRs. As depicted in Figures 8 and 9, 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™or the Breakout Board, and can then be connected to other modules through interfaces.

B. Interface Definitions

After defining the modules, the generic interfaces must be defined. The interface definition collection is centralized in

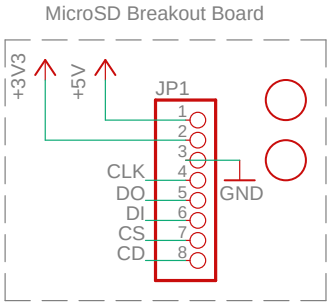

Figure 7. Schematics of a placeholder design block for a MicroSD Breakout Board [29].

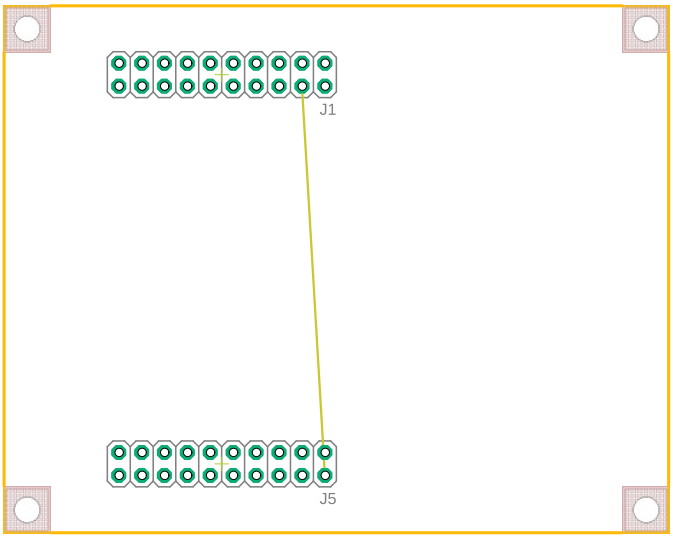

Figure 8. Board layout of a placeholder design block for a TI LaunchPad™ [27].

a single file, and its structure is shown in Figure 10. In this example, only SPI and I²C have been defined with its standard connections. As the format is generic, other interface types, e.g. Controller Area Network (CAN, [32]) or even Advanced eXtensible Interface Bus (AXI, [33]), 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 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 14, Figure 10), whereas a master has as many CS wires as it has slaves connected to it (marked with *wiremultiple*; cf. line 13, Figure 10). Compared to SPI, the shown example of I²C is rather simple, as it only consists of two wires, with a master/slave concept as well. All participants are simply connected to the corresponding bus wires.

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 11. 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™ *MSP1* was connected to a MicroSD Breakout Board *SD1* via SPI. This particular SPI connection is called *SPI_Connection1* of type *SPI* and has two participants with different roles: *MSP1* as

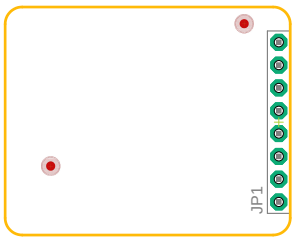

Figure 9. Board layout of a placeholder design block for a MicroSD Breakout Board [29].

```
{
  interfaces: [
    {
      type: "SPI",
      connections: [
        { "master.MOSI" : "bus.MOSI" },
        { "master.MISO" : "bus.MISO" },
        { "master.SCLK" : "bus.SCLK" },
        { "slave.MOSI"  : "bus.MOSI" },
        { "slave.MISO"  : "bus.MISO" },
        { "slave.SCLK"  : "bus.SCLK" },

        { "master.CS"   : "wiremultiple" },
        { "slave.CS"    : "wiresingle" }
      ]
    },
    {
      type: "I2C",
      connections: [
        { "master.SDA" : "bus.SDA" },
        { "master.SCL" : "bus.SCL" },
        { "slave.SDA"  : "bus.SDA" },
        { "slave.SCL"  : "bus.SCL" }
      ]
    }
  ]
}
```

Figure 10. Interface definition containing SPI and I^2C.

a master and *SD1* as a slave. This system definition will prospectively be generated and extracted out of the ASW code by the analysis step in ***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 subsequently (in part manually). 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.

```
{
  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 11. A system model containing two modules connected via SPI.

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 4). These pins must, therefore, be matched with the connection's wires (cf. Figure 10). 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. The basis of every schematic and board plan forms an empty EAGLE plan, on which the explained actions are performed.

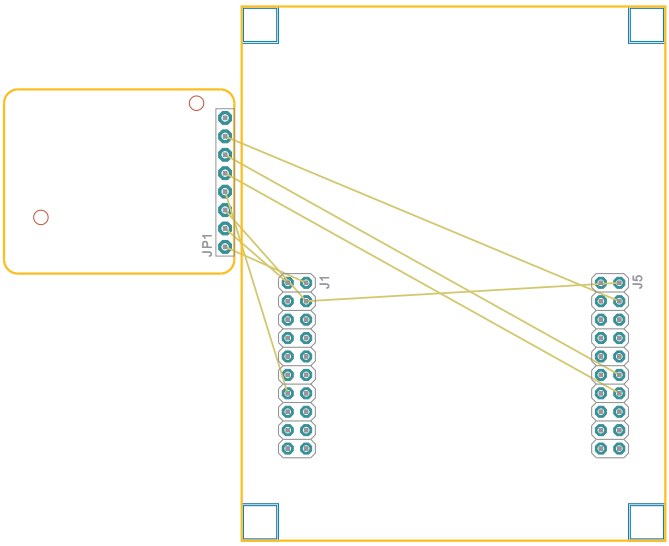

Figure 12. Raw output of the board layout generated as displayed in EAGLE.

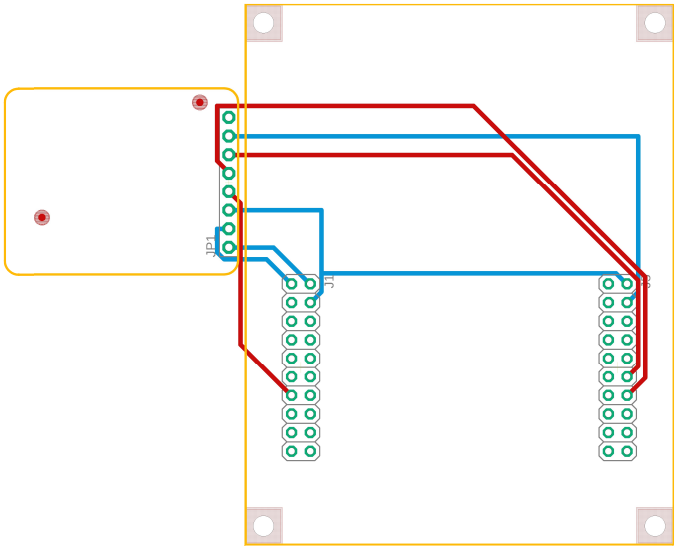

Figure 13. Board layout after auto-routing in EAGLE.

(1) **Instantiation:** In this step, each module must be instantiated by loading the corresponding design block of its type. To avoid overlapping of signal names and, therefore, unwanted connections between modules of the same type, suffixes are added according to the instance's name. For readability purposes, these suffixes are equal to the instance's name defined in the system definition file (e.g., **_MSP1*). This can be easily seen when comparing, e.g., Figures 6/7 and 14.

(2) **Interconnection:** This step must be 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. In this case, to avoid overlapping of signals, a dot notation style is used do distinguish between wires of different connection instances (e.g., *SPI_CONNECTION1.** in Figure 14).

(3) **Placement:** 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 (one for the schematics, one for the board layout) 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. Experiments and Evaluation

Within this section, the previously explained concept on how to define and create PCBs from a definition language is shown in different examples and evaluation. At first, a simple proof of concept is presented in Section VI-A, followed by some analysis and evaluation on scalability and performance of the algorithms in Section VI-B. Section VI-C shows a use case with a corresponding manufactured and equipped prototype PCB. In this case, a comparison with other approaches is not executed, as all related works go in different directions. Hence, there are no acceptable metrics for comparison provided.

A. Proof of Concept

The proof of concept comprises the generation of the system definition as shown in Figure 11. As mentioned above, the system created consists of two modules interconnected with one SPI bus, whereas the processor board serves as master. The schematics generation step yields in the drawing depicted in Figure 14. Compared with the LaunchPad™'s design block shown in Figure 6, 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 is given 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 12, as described in Section V-B. The fine lines show non-routed connections between the pins. As the generated plan will, of course, be

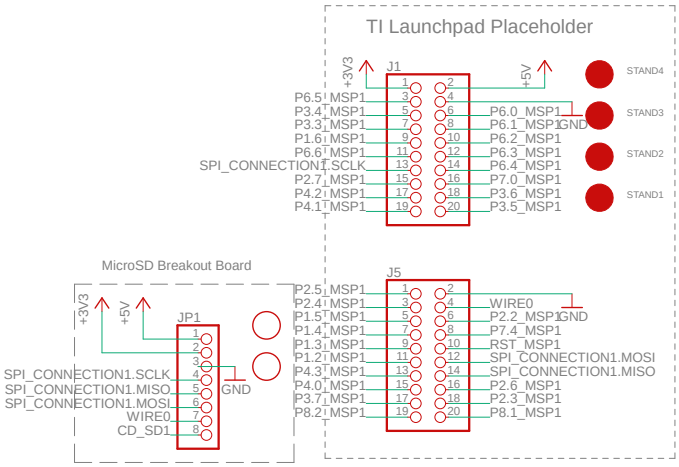

Figure 14. Raw output of the schematics generated as displayed in EAGLE.

manufactured as a real hardware PCB, no single component can overlap in the final layout. 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 13. EAGLE can also be used to check the correctness of the XML file format.

B. 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 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. The experiment is based on the one presented in [1], but the generation tool version is more stable and optimized and the objective differs slightly, yielding different measurements.

Each test case is based on the example described in Section VI-A, 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 seven 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^2C connection and was similar to scenario one, whereas scenario four included SPI and I^2C 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 15. When comparing all scenarios, the trend observed is relatively similar: All performance graphs show a linear devolution with an additive, logarithmic-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

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)	656.22 ms	94 KiB
1	3 part. (1 M, 2 S)	751.98 ms	111 KiB
2	4 part. (1 M, 3 S)	852.95 ms	128 KiB
3	5 part. (1 M, 4 S)	933.71 ms	144 KiB
4	6 part. (1 M, 5 S)	1 013.67 ms	161 KiB
5	7 part. (1 M, 6 S)	1 108.81 ms	178 KiB
6	7 part. (1 M, 7 S)	1 193.25 ms	194 KiB
	2 SPI conn. (scenario 2)		
0	4 part. (1 M and 1 S each)	890.77 ms	160 KiB
1	6 part. (1 M and 2 S each)	1 060.91 ms	192 KiB
2	8 part. (1 M and 3 S each)	1 234.36 ms	226 KiB
3	10 part. (1 M and 4 S each)	1 398.30 ms	259 KiB
4	12 part. (1 M and 5 S each)	1 557.30 ms	293 KiB
5	14 part. (1 M and 6 S each)	1 731.26 ms	326 KiB
6	14 part. (1 M and 7 S each)	1 879.93 ms	360 KiB
	1 I^2C conn. (scenario 3)		
0	2 part. (1 M, 1 S)	665.61 ms	106 KiB
1	3 part. (1 M, 2 S)	771.77 ms	136 KiB
2	4 part. (1 M, 3 S)	889.56 ms	167 KiB
3	5 part. (1 M, 4 S)	989.33 ms	198 KiB
4	6 part. (1 M, 5 S)	1 106.84 ms	228 KiB
5	7 part. (1 M, 6 S)	1 205.48 ms	259 KiB
6	7 part. (1 M, 7 S)	1 332.82 ms	290 KiB
	1 I^2C and 1 SPI conn. (scenario 4)		
0	3 part. (1 M, 1 S each)	782.70 ms	127 KiB
1	5 part. (1 M, 2 S each)	971.42 ms	174 KiB
2	7 part. (1 M, 3 S each)	1 166.01 ms	222 KiB
3	9 part. (1 M, 4 S each)	1 344.08 ms	269 KiB
4	11 part. (1 M, 5 S each)	1 530.92 ms	317 KiB
5	13 part. (1 M, 6 S each)	1 719.09 ms	364 KiB
6	13 part. (1 M, 7 S each)	1 873.92 ms	411 KiB

doubling the numbers in the first test case resulted 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.

C. Simple Use Case and Prototype Manufacturing

The simple use case in this section is a minimalistic, generic control system. In order to react to its environment, it must

- read several analog voltage values,
- output analog voltage values, and
- store a data log permanently in two different ways, such as it is on the one hand side
  - "detachable", and on the other hand side
  - stored with low energy consumption, yet non-volatile and redundant.

After manually spanning a selection space over the available equipment in our lab, those requirements yield in a system configuration with

- an MCU to execute the control algorithms ($\rightarrow$ MSP430 on a corresponding LaunchPad™),
- two 4-channel analog-to-digital converters (ADCs) to read voltage values ($\rightarrow$ Adafruit 4-channel Breakout Board featuring an ADS1115 ADC [34]),
- a digital-to-analog converter (DAC) to output voltage values ($\rightarrow$ Adafruit 12-bit DAC board featuring a MCP4725 DAC [35]),

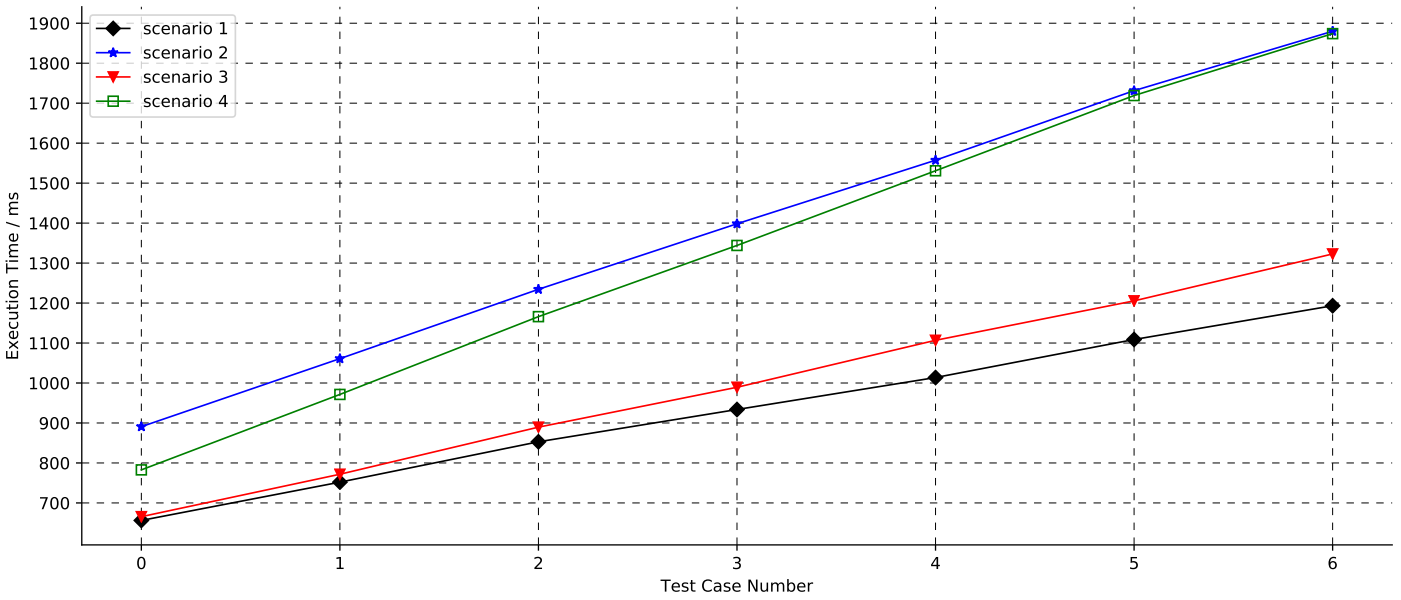

Figure 15. Performance graph for different test cases in all four scenarios.

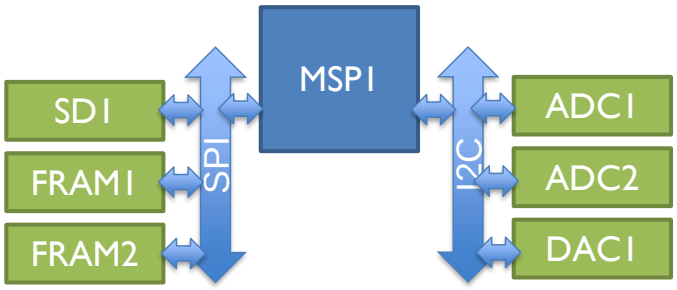

Figure 16. The block diagram of all modules in the example use case.

- a MicroSD card module to log data in a detachable way (→ MicroSD Breakout Board), and
- two Ferroelectric Random Access Memory (FRAM, [36]) modules to store data in a low-power, non-volatile, redundant way (→ Adafruit SPI FRAM Breakout Board featuring a MB85RS64V FRAM [37]).

The block diagram of this configuration, including its interconnection, is shown in Figure 16. It can be seen that a total of two different connection types must be used to interconnect all modules. The corresponding system definition is presented in Figure 17. It contains all module instances, both connections with their types, participants, and roles. The result of running the PCB generation and manually routing the board layout is depicted in Figure 18. With this result it is possible to manufacture an actual PCB, equip it with the hardware modules, flash the control system ASW and BSW, and run measurements. The software setup in this case consists of our own real-time operating system *MCSmart*OS [38][39] enriched with a modular driver management system and a simple test application. The equipped and running prototype is shown in Figure 19, where it is connected to several measurement devices (e.g., a PicoScope 2205 MSO [40] with digital and analog inputs) through debug wires and probes to observe and verify correct functionality.

```
{
  modules: [
    { name: "MSP1",  type: "MSP430F5529_Launchpad" },
    { name: "ADC1",
      type: "Adafruit_ADS1115_16Bit_I2C_ADC" },
    { name: "ADC2",
      type: "Adafruit_ADS1115_16Bit_I2C_ADC" },
    { name: "DAC1",
      type: "Adafruit_MCP4725_12Bit_I2C_DAC" },
    { name: "FRAM1", type: "Adafruit_FRAM_SPI" },
    { name: "FRAM2", type: "Adafruit_FRAM_SPI" },
    { name: "SD1",   type: "MicroSD_BreakoutBoard" }
  ],

  connections: [
    {
      name: "I2C_Connection1",
      type: "I2C",
      participants: [
        { name: "MSP1", role: "master" },
        { name: "ADC1", role: "slave" },
        { name: "ADC2", role: "slave" },
        { name: "DAC1", role: "slave" }
      ]
    },
    {
      name: "SPI_Connection1",
      type: "SPI",
      participants: [
        { name: "MSP1",  role: "master" },
        { name: "FRAM1", role: "slave" },
        { name: "FRAM2", role: "slave" },
        { name: "SD1",   role: "slave" }
      ]
    }
  ]
}
```

Figure 17. The system model of the prototype.

VII. Conclusion and Future Work

In conclusion, the present work based on the ***papageno***PCB approach represents a novel, top-down concept to develop an embedded system for a multitude of possible application scopes. Having only 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 numerous new possibilities towards automatic system generation

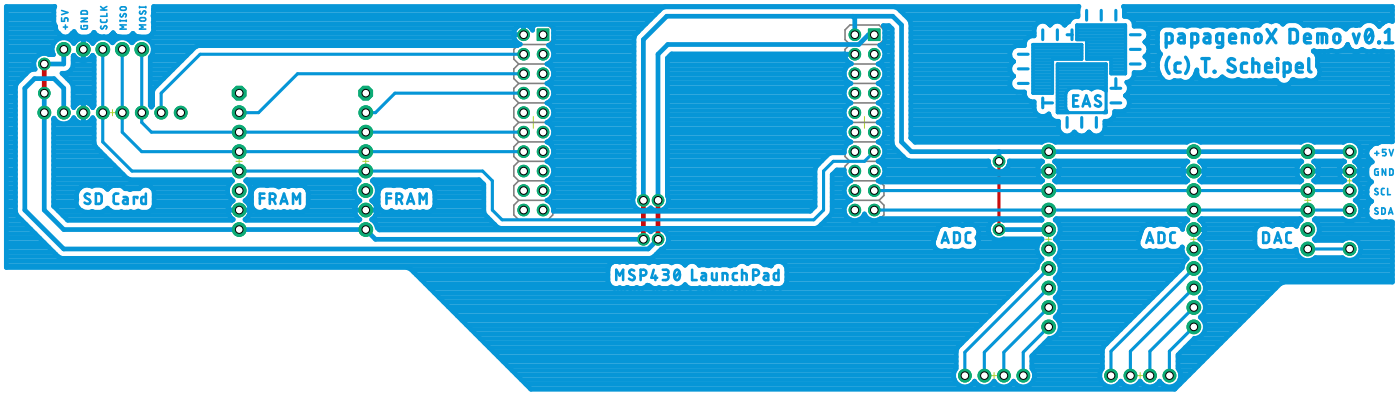

Figure 18. Prototype PCB layout with a MSP430 LaunchPad™ connected to three SPI and three I²C modules (manually routed).

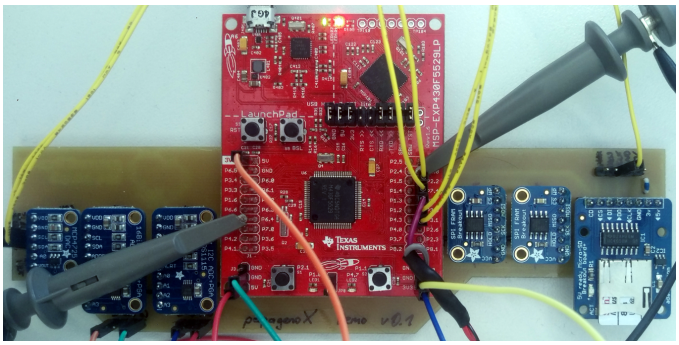

Figure 19. Fully equipped prototype board with debug wires.

and across several abstraction levels including, e.g., automatic bus balancing, bandwidth engineering, optimization towards functional and non-functional hardware requirements. All these things can be carried out even before building the actual hardware for the system. 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 in terms of non-functional metrics such as bandwidth or power consumption. Generally speaking, the presented concept is able to optimize systems under development regarding different, user-defined metrics already at design level. Therefore, metrics to measure the overall improvement in general are hard to define, as they depend on the actual system's development process and its requirements and properties. 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 electrical device or component level, proper definitions presumed.

Concerning future work, a detailed extraction of system models from a profound ASW source code analysis is of utmost importance. Therefore, we are working on introducing annotations into our operating system environment [39], 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 done to improve the automatic portability of real-time operating systems [41], 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 [42] or application-specific logic components on reconfigurable computing platforms could be created and included by taking this approach. The ultimate goal is to establish ***papageno***X as a universal embedded systems generator, which uses only ASW source code or models as an input.

References

[1] T. Scheipel and M. Baunach, "papagenoPCB: An Automated Printed Circuit Board Generation Approach for Embedded Systems Prototyping," in *ICONS 2019 - The Fourteenth International Conference on Systems*, 3 2019, pp. 20–25.

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

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

[4] A. Kouba, J. Navratil, and B. Hnilička, "Engine Control using a Real-Time 1D Engine Model," in *VPC – Simulation und Test 2015*, J. Liebl and C. Beidl, Eds. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, pp. 295–309.

[5] Infineon Technologies AG, "TC1797 – 32-Bit Single-Chip Microcontroller," 2014.

[6] B. Eichberger, E. Unger, and M. Oswald, "Design of a versatile rapid prototyping engine management system," in *Proceedings of the FISITA 2012 World Automotive Congress*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 135–142.

[7] Infineon Technologies AG, "TC1796 – 32-Bit Single-Chip Microcontroller," 2007.

[8] ——, "TC1798 – 32-Bit Single-Chip Microcontroller," 2014.

[9] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. UK: Strathclyde Academic Media, 2014.

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

[11] 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.

[12] B. Singh, D. O'Riordan, B. G. Arsintescu, A. Goel, and D. R. Deshpande, "System and method for circuit schematic generation," US Patent US7 917 877B2, 2011.

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

[14] B. Schommer, C. Cullmann, G. Gebhard, X. Leroy, M. Schmidt, and S. Wegener, "Embedded Program Annotations for WCET Analysis," in *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*. Barcelona, Spain: Dagstuhl Publishing, Jul. 2018, [retrieved: Nov, 2019]. [Online]. Available: https://hal.inria.fr/hal-01848686

[15] S. Chakravarty, Z. Zhao, and A. Gerstlauer, "Automated, retargetable back-annotation for host compiled performance and power modeling," in *9th Int'l Conference on Hardware/Software Codesign and System Synthesis*, Piscataway, NJ, USA, 2013, pp. 36:1–36:10, [retrieved: Nov, 2019]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555692.2555728

[16] A. D. Pimentel, "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration," *IEEE Design Test*, vol. 34, no. 1, pp. 77–90, Feb 2017.

[17] F. Herrera, H. Posadas, P. Peñil, E. Villar, F. Ferrero, R. Valencia, and G. Palermo, "The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 55 – 78, 2014, [retrieved: Nov, 2019]. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S138376211300194X

[18] T. Saxena and G. Karsai, "A meta-framework for design space exploration," in *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, April 2011, pp. 71–80.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543.

[20] AUTOSAR, "Classic platform release 4.3.1," 2017.

[21] IEEE Standards Association, *IEEE 1076-2008 - IEEE Standard VHDL Language Reference Manual*, 2008.

[22] ——, *IEEE 1364-2005 - IEEE Standard for Verilog Hardware Description Language*, 2005.

[23] ——, *IEEE 1666-2011 - IEEE Standard for Standard SystemC Language*, Sep. 2011.

[24] B. Maes, "Comparison of contemporary file systems," *Citeseer*, 2012.

[25] S. C. Hill, J. Jelemensky, M. R. Heene, S. E. Groves, and D. N. Debrito, "Queued serial peripheral interface for use in a data processing system," US Patent US4 816 996, 1989.

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

[27] Texas Instruments, *MSP430F5529 LaunchPad™ Development Kit (MSP--EXP430F5529LP)*, Apr. 2017.

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

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

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

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

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

[33] ARM Ltd., *AMBA AXI and ACE Protocol Specification*, 2017, [retrieved: Jul, 2019].

[34] Texas Instruments, *Ultra-Small, Low-Power,16-Bit Analog-to-Digital Converter with Internal Reference*, Oct. 2009, [retrieved: Nov, 2019]. [Online]. Available: http://www.ti.com/lit/ds/symlink/ads1114.pdf

[35] Microchip, *12-Bit Digital-to-Analog Converter with EEPROM Memory in SOT-23-6*, 2009, [retrieved: Nov, 2019]. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/mcp4725.pdf

[36] H. Ishiwara, M. Okuyama, and Y. Arimoto, *Ferroelectric random access memories: fundamentals and applications*. Springer Science & Business Media, 2004, vol. 93.

[37] Fujitsu Semiconductor, *64KBit SPIMB85RS64V*, 2013, [retrieved: Nov, 2019]. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/MB85RS64V-DS501-00015-4v0-E.pdf

[38] M. Baunach, "Advances in Distributed Real-Time Sensor/Actuator Systems Operation," Dissertation, University of Würzburg, Germany, Feb. 2013. [Online]. Available: http://opus.bibliothek.uni-wuerzburg.de/frontdoor/index/index/docId/6429

[39] 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.

[40] *PicoScope 2205 MSO Mixed Signal Oscilloscope*, Pico Technology, 2016. [Online]. Available: https://www.picotech.com/download/datasheets/PicoScope2205MSODatasheet-en.pdf

[41] 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.

[42] 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.