Received June 22, 2021, accepted July 6, 2021, date of publication July 12, 2021, date of current version August 3, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3096749

Micraspis: A Computer-Aided Proposal Toward Programming and Architecting Smart IoT Wearables

LONG-PHUOC TÔN[1,2,4], LAM-SON LÊ[1,2], (Member, IEEE), AND MINH-SON NGUYEN[2,3]

[1]Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Ho Chi Minh 7000, Vietnam
[2]Vietnam National University–Ho Chi Minh City, Thu Duc 70885, Vietnam
[3]Faculty of Computer Engineering, University of Information Technology (UIT), Thu Duc 70885, Vietnam
[4]Faculty of Information Technology, Industrial University of Ho Chi Minh City, Ho Chi Minh 7000, Vietnam

Corresponding author: Lam-Son Lê (lam-son.le@alumni.epfl.ch)

ABSTRACT A wearable is a lightweight body-worn device that relies on data-driven communications to keep people connected purposefully, for instance, for fire-fighting, prompting fast-food clients, and medical treatment. With rise of wearable computing in the era of IoT-driven smart applications, programmers now expect the time to market for these devices to be shortened. While support for IoT programming in general has gathered traction, tool proposals that automate the development of smart solutions based on the Internet of Wearable Things, though of paramount importance, still stay on the sidelines. We propose a code generation tool called Micraspis that allows a wearable to be described both functionally and architecturally – as if they are two sides of the same coin. The tool has an underlying model-to-code transformation mechanism to generates source code that is executable on a specific IoT programming platform such as Arduino. Our experiments demonstrate that programming code generated by Micraspis amounts to at least 60% of the source code needed to fulfill the business logic of ordinary wearable devices. We conduct an interview to meticulously collect programmers' assessment on how Micraspis assists them in programming and architecting smart IoT wearables. A total of 161 programmers responded to a Likert scale questionnaire, with which at least 65% of them either agree or strongly agree. Overall, the results show that Micraspis has promising applicability in supporting IoWT-enabled smart solutions.

INDEX TERMS Wearable computing, code generation, state machine, Internet of Wearable Things.

I. INTRODUCTION

Following the Internet of Things (IoT) as as paradigm shift in recent times [1], the proliferation of small body-worn devices – known as IoT wearables or simply wearables, has attracted the attention of business analysts, solution architects and system integrators alike. Along with the explosion of digital transformation, a new segment came up as the Internet of Wearable Things (IoWT) [2]–[4]. A wearable is a lightweight body-worn personal device that is in use for specific purposes, e.g., monitoring wearer's health condition, coordinating rescue work at a disaster site [5]. Owing to the ever-expanding capabilities of wearable sensors and wearable technologies, IoWT promises to create an uptake of personal healthcare solutions [5]–[8], agriculture production [9] and smart eyewear [10]. Uses of wearable devices continue to keep pace with IoT technological advancement [11]. Despite their rather limited computing power, wearable devices play a central role in many smart applications, thanks to their mobility and connectivity [4], [12]. This shift in computing paradigm would unlock the potential for the next-generation software architecture, analogously to how the Web-based software development communities enjoyed the mainstream n-tier architecture that was made popular in the last decade [13]. Business processes operated in such an IoWT-enabled setting should be geared up to best harvest the mobility and IoT data-driven connectivity to keep people connected purposefully for firefighting, collecting fast-food orders, medical treatment, to name just a few [14], [15].

There are three fundamental yet separable functions of an IoT-driven application: capturing raw data from the devices, transmitting information extracted from the data over a designated network, and taking action based on progressively built-up insights. From a software engineering's point

The associate editor coordinating the review of this manuscript and approving it for publication was Adnan M. Abu-Mahfouz.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

of view, the enactment of business processes in such an IoT-based solution necessitates, to some degree, the automation of these functions [16], [17] though they could be engineered separately. It is noted that universal theories and tools for developing software may not come into existence [18]. General consensus in this discipline is that software engineers should narrow down the business domain to gain automation on certain software activities [19]. The business domain of an IoT-enabled smart application is characterized by a set of business processes being enacted, where heterogeneous IoT devices take part in autonomously. Let's take smart firefighting rescue as an example. Firefighters wear a designated wearable device that allows them to report victims they spot to a coordinating team by communicating with an on-site server. They also receive instructions from the same server of whether they should take care of the victim they found or keep searching for unspotted ones, leaving the spotted victim for a better-skilled rescuer to come and perform a professional first aid. The rescuer's wearable takes part in at least two business processes: reporting victims and performing rescue. The entire solution for this on-site firefighting rescue is essentially a heterogeneous system that involves both contemporary server-side programming and system programming (for the rescuer's wearable). Heterogeneity makes the first challenge when it comes to software automation for these smart applications [16], [20], [21]. Another challenge is that while the business processes are subject to change, one may expect time for having them enacted to be shortened [22]. Let's take the pandemic situation of Covid-19 as another example where many organizations and public healthcare institutions demand a symptom-checking application in a rather short time-frame. Business processes enacted using this symptom checker to scan the skin temperature of big crowds are prone to change since they depend on often-revised health measures and regulation exercised by the health ministry.

General underpinning for IoT programming ranges from access control [23], the model-view-controller pattern [24], logic programming [25], script writing [26], flow-based programming [27], [28], to visual programming [29], [30]. Model-based engineering in this realm either target IoT heterogeneous objects [16], [31], mission-critical IoT systems [32], architecture and concepts [33] or propose an extension to proven object-orientation standards to cover certain IoT domains [17], [34]–[36]. Nevertheless, domain-specific approaches to IoT-enabled software architecture leave a lot to be desired [37]. When it comes to code generation engines for automating the development of IoT applications, existing scholar work looks into ontology [38], state machine [39], code integration [40], and specific IoT platforms [41]. The construction of a wearable device (and therefore the enactment of the business processes it takes part in) involves not only a component-assembly design (i.e., wiring commonly available hardware components to a mainboard) but also some degree of hardware programming – just as two sides of the same coin.

Proposals towards model-driven engineering and model-to-code transformation for IoT-based software architecture continue to catch up. The mainstream thought in this realm is to raise the abstraction level with the objective of accelerating the software development process and to enable design and code reuse [21]. Unfortunately, direct support for IoWT programming still stays on the sideline – to the best of our understanding, no tool proposals have specifically been devised for. Existing work on IoWT tends to discuss the research agenda for wearable computing in a rather broad scope [8], [42]–[45]. Computer-aided solutions to the construction of IoT devices leave a lot to be desired. To construct a wearable device that will be functionally ready for a business process, we shall (i) propose an effective component-assembly design using common hardware components (e.g., LED, buzzer, keypad); (ii) program the chosen hardware components to realize the black-box specification of the wearable in question in accordance to the said business process. Our first attempt in this research line has resulted in our early thoughts on a domain-specific modeling plus a prototypical code generation engine [46], and a proposal for semi-automatically enacting business processes in smart healthcare [47]. In this article, we present our tool called Micraspis, which follows a model-to-code transformation approach and fulfills the aforementioned requirements for the sake of enacting business processes where IoWT plays a central role. We investigate the notion of well-formedness on the design of a wearable as a whole. We analyze the co-existence of and the correlation between its architectural design and behavioral spec, which together with the design well-formedness, have resulted in a definition of hardware constraints and syntactic check used for validating a wearable design. Furthermore, we study the roles such a tool might play in a rather extended lifecycle of wearable devices. Our tool (a) offers a visual hardware design for architecting the wearable device to be constructed; (b) allows the business logic of a wearable device to be represented primarily in the form of a state machine – a proven way of effectively capturing the behavior of IoT devices and objects [48]–[51]; (c) generates source code and auxiliary files needed to program and deploy the wearable device in question. We discuss the applicability of Micraspis through a series of mini-projects with a logical progression to complexity. To validate our computer-aided proposal, we measure the completeness of the generated source code and the usefulness of the tool from the programmers' perspective.

Paper Structure. Section II presents our research motivation and formulates our research questions. Related work is surveyed in Section III. Section IV describes a layered architecture concerning our code generation proposal and presents how our Micraspis tool contributes to the software automation in wearable computing. Section V reports experiments conducted on our tool proposal. Section VI draws some concluding remarks and points out the future work.

II. PROBLEM STATEMENT

A. CASE STUDY

Epidemics of dengue and measles often occur in tropical countries like Vietnam.[1] Effective healthcare for and treatment of these two infectious diseases is crucial to containing an outbreak. When a large number of patients are hospitalized in an outbreak, hospital doctors struggle to deliver medical care, especially when their patients are mixed with children. Traditionally, doctors rely on their nurses to keep an eye on the development of these diseases using medical thermometers. To handle this stressful situation, they finally resort to an IoT-based system called *iTempFoll* that helps them monitor and treat their patients with minimal delay. There are two types of wearable being put in use for *iTempFoll*: the patients wear one of them and the other is worn by doctors or nurses. These body-worn devices, often of compact size, are coordinated by a server that collects and analyzes patients' body temperatures as time-series data. The server hosts a module allowing the hospital management to define medical rules for effectively tackling dengue and measles outbreaks in the hospital. These rules, combined with measurement data collected from the patients, enable the server to decide what treatment should be applied, which otherwise has to be decided by the struggling doctors in a chaotic hospital environment.

What we learn from the coronavirus epidemic happening in early 2020 into the beginning of 2021 is analyzing the fluctuation of body temperature of infected patients is of paramount importance. According to the Centers for Disease Control and Prevention, patients infected with Covid-19 need to stay home in isolation and be monitored for symptoms such as fever, cough, and others.[2] Because of that oddity, *iTempFoll* would set an example that finds increasing applicability in the pandemic time. The coordinating server keeps track of the historical body temperature measurement, allowing a more insightful medical analysis to be done – as opposed to a quick, superficial check of the instant body temperature measurement (unfortunately, this is the only measure typically implemented by a Covid-19 checkpoint at the entrance of many buildings upon the arrival of a new wave of coronavirus).

A solution architecture for *iTempFoll* is depicted in Figure 1. Speaking of the hardware architecture, the patient's wearable is made of an Arduino board and a couple of components, including a body temperature sensor and LEDs that show the wearing patient's current status. Likewise, a doctor's wearable consists of an Arduino board, a keypad, and an LCD for a higher degree of human-device interaction (e.g., browsing and viewing detailed information about the patients the wearing doctor takes care of). Both of these wearables can transmit information to the coordinating server via built-in Wifi that is readily available

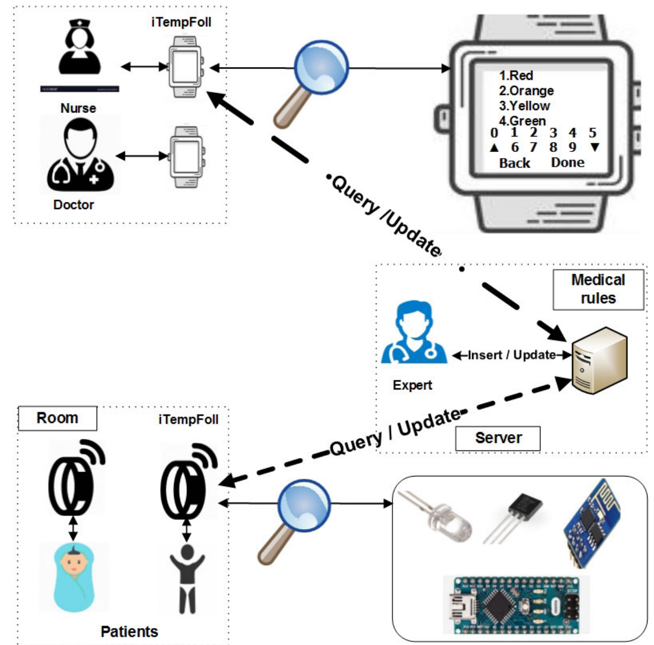


FIGURE 1. IoT-driven architecture of *iTempFoll*.

on the said Arduino board. Programming these hardware components to meet medical rules necessary for handling the dengue/measles outbreak and perhaps the coronavirus situation is typically a time-consuming task. Even when programmed and properly bootloaded,[3] they are still subject to fine-tuning and re-programming due to changes in medical policies and procedures, especially when rushing to contain a new outbreak, necessitating code generation techniques that ease the programming of wearable devices. Finding an effective and systematic way of describing the business logic of those wearable computers is essential for making such code generation possible by, for example, harvesting some proven model transformation techniques.

The first wearable of *iTempFoll* is for, and supposed to be worn by, patients. It periodically sends the wearer's real-time body temperature to the coordinating server. Many of them are in a serious health condition and may not be able to cooperate voluntarily. The server keeps track of the patient's body temperature and decides what to do next in a rule-based manner. For example, as listed in the row about dengue in Table 1, if the patient's body temperature falls between 37.5 and 38.5 degree Celsius for over 10 minutes and less than 48 hours, the patient's wearable will turn yellow to signify an incubation phase. Simultaneously, the coordinating server activates the said medical rule to notify a doctor/nurse responsible for this patient. She is then supposed to perform an X-ray for this patient and watch for other symptoms. A server-side rule activation also results in a confirmation being sent back to the patient's wearable. This

[1] According to the National Agency of Preventive Medicine, Vietnam has between 50 and 150 thousand cases of dengue every year. Particularly in 2017, Vietnam had 183,287 dengue cases, of which 30 died.

[2] What to Do If You Are Sick https://www.cdc.gov/coronavirus/2019-ncov/if-you-are-sick/steps-when-sick.html. Accessed 10 July 2020

[3] Microcontrollers are usually programmed through a bootloader unless it comes with an alternative piece of firmware that allows directly installing new firmware.

TABLE 1. Medical rules for taking care of patients showing symptoms of dengue and measles.

Disease	Condition	Illumination	Treatment
Dengue	temp ∈ [37.5, 38.5] & time ∈ [0.15, 48.0]	Yellow	Incubation phase (chest x-ray and abdominal ultrasound)
	temp ∈ [39.0, 40.0] & time ∈ [0.15, 23.99]	Orange	Febrile phase (monitoring for warning signs and other clinical parameters)
	temp ∈ [39.0, 40.0] & time ∈ [24.0, 48.0]	Red	Emergency phase (measure hematocrit every 1-2 hours, once every 6 hours)
	temp ∈ [37.0, 37.5] & time ∈ [0.15, 48.0]	Green	Recovery phase (close monitoring is necessary to recognize heart failure or pulmonary edema)
	in emergency or dangerous	Red/Orange	Emergency phase (call and text doctors / nursers right away)
Measles	temp ∈ [37.5, 38.5] & time ∈ [0.15, 48.0]	Yellow	Incubation phase (e.g. blood test, x-rays)
	temp ∈ [38.5, 40.0] & time ∈ [0.15, 23.99]	Orange	Febrile phase (antipyretic, serological tests)
	temp ∈ [38.5, 40.0] & time ∈ [24.0, 48.0]	Red	Emergency phase (antipyretic, supportive therapy)
	temp ∈ [37.0, 37.5] & time ∈ [0.15, 48.0]	Green	Recovery phase (close monitoring is necessary to recognize heart failure)

update of the patient's medical condition is illuminated in colors telling how critical the wearer's medical condition is in.

The second wearable of *iTempFoll* is to be worn by the medical staff, i.e., doctors and nurses. Its wearer can query the status of any patient she is responsible for. In case a medical rule kicks in, this wearable receives an alert sent from the coordinating server. The server maintains a rather simple database assigning a doctor or nurse to patients she takes care of in the hospital. The entire medical process of monitoring patients and alerting doctors is described in Figure 2 using a de-facto modeling standard.

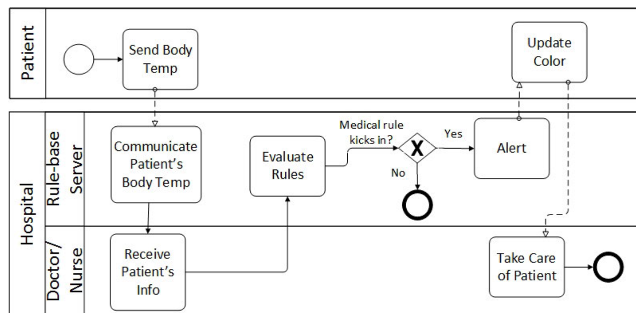


FIGURE 2. The medical process of *iTempFoll*.

B. RESEARCH QUESTIONS

The rationale behind model-driven engineering approaches is that the implementation should best be derived from its model through a systematic model-to-code transformation. To be able to build a code generation tool for wearable devices, we first need to ground some fundamental research questions.

1) RQ1: DUAL-FACETED VISUAL DESIGN

Hardware design for embedded systems and IoT-based smart systems alike has long been associated with an intuitive assembly-component way of architecting. Software design for these devices should follow suit. Furthermore, the two facets of design should be coupled semantically, just as they are two sides of the same coin.

2) RQ2: SEPARATION OF CONCERNS FOR THE SAKE OF MODEL TRANSFORMATION

On the one hand, software abstraction and hardware design of the same wearable computer could be done separately in light of the so-called principle of separation of concerns. On the other hand, to enable model-to-code transformation, design of the wearable as a whole should be cohesive and coherent, i.e. the behavioral description needs to be semantically linked to its hardware counterpart of the same design. More concretely, if we adopt UML as a de facto language for capturing the wearable's state machine, we would better formulate the operations and transitions populated in this diagram in terms of the hardware components used for architecting the wearable in question.

3) RQ3: CODE GENERATION THAT REALIZES THE STATE MACHINE AND THE PIN CONFIGURATION OF THE WEARABLE TO BE CONSTRUCTED

Programmatically, the state machine of devices in a IoT/embedded system is primarily implemented as a nested switch statement in C programming [52]. We need a model-to-code transformation that effectively yields this C programming construct out of a well-formed UML statechart diagram – a widely practiced apparatus for modeling the state machine. Yet another programming burden when implementing a wearable computer is to make sure the pin variables representing the chosen hardware components (say, in C source code) match how they are plugged physically into its mainboard. Both of these coding exercises are burdensome and error-prone, necessitating a dedicated code generation tool.

III. RELATED WORK

This section is dedicated to the state-of-the-art of the programming support for IoT-based architectures, which is not necessarily confined to IoWT. In Subsection III-A, we discuss the code generation techniques used for IoT devices and embedded systems. In Subsection III-B, we look into model-engineering proposals for IoT programming.

A. CODE GENERATION BASED ON STATE MACHINE

State machine is a conventional schema used to describe the behavior of devices and objects in system modeling.

Generating source code from the state machine has been entertained in many model transformation approaches [52].[4] In general, source code generated from the state machine, be it a UML statechart, finite state machine, or Harel state chart, is diverse in terms of programming languages. In particular, generating source code for IoT applications from the state machine has been studied extensively [53], and the results look encouraging. Despite sounding promising, this work faces the most prominent challenge that is the incapability of reusing the generated source code on different devices having the same hardware architecture.

B. MODEL-DRIVEN ENGINEERING FOR IoT PROGRAMMING

Model-driven engineering (MDE) has long been advocated by scholars and practitioners in software engineering. A great deal of effort has been put into developing high-level domain-specific languages (DSL) since IoT programming came into the stage [54]. The rationale behind most DSLs is to abstract the application logic and peculiarities into interconnected blocks via library modules. The general consensus for engineering a DSL is that business analysts and requirements engineers alike wish to raise its level of abstraction to stay focused on the business logic, leaving the burden of obtaining functional programming code to its underlying code generation engine [19]. MDE has started to draw traction in the realm of IoT-enabled smart applications soon after the IoT as a computing paradigm shift gave rise to the next generation of software architectures and practices. In this subsection, we offer an analysis of existing MDE frameworks for IoT programming and assess them in light of our problem statement given in Section II.

1) MIDGAR

Midgar [55] is an IoT programming platform that facilitates code generation. Midgar allows a system specification to be abstracted in a DSL via a Web-based editor. Source code generated by Midgar is deployable on interconnected heterogeneous objects that were pre-registered and pre-defined. Code generation is performed through a series of six nodes. The first four nodes correspond to the programming language structure, including one conditional node, two-loop nodes, and one sleeping node. The fifth node allows for direct code insertion, and the last one describes the actions to be executed. Midgar neither supports the modeling of state machines nor the architectural design of a wearable.

2) Asm2C++

Asm2C++ is a tool that automatically generates executable C++ code for Arduino from a formal specification [39] articulated as state machines. The code generation process of Asm2C++, which is part of a broader framework for the analysis and validation of the model correctness [56], follows a model-driven engineering approach where certain transformation rules are applied to transform an abstract state machines into executable code [57]. The tool does support the modeling of state machines and hardware internals of an IoT device. However, the principle of separation of concerns is not fully respected due to high coupling between these dual design facets.

3) IoTSuite

IoTSuite is an IoT application development platform that allows an IoT-based system to be developed through a series of design, implementation, and deployment [40]. By having a compiler and deployment modules integrated, various stages of development could be automated in IoTSuite. The platform also allows for the integration of a modeling language that make highly abstract models expressible. Its support for hardware design is rather limited. Unfortunately, the source code generated by this platform does not satisfactorily implements low-level event-driven handlers commonly found in an IoT-based system.

4) UML4IoT

UML4IoT is a UML-based approach to exploit MDE in the development of IoT systems [17]. By proposing a designated UML profile, this approach establishes an IoT-compliant layer into which the cyber-physical components of an IoT system are transformed. In order for this wrapper layer to generate what is essentially required for the cyber-physical components to be effectively integrated into the modern IoT production environment, UML4IoT looks into both the UML diagrams (mostly activity diagrams) and the embedded components of an IoT system. Code generation for what is dubbed IoTwrapper is the primary contribution of this approach towards the transformation of legacy systems into an Industry 4.0 ready environment.

5) VIPLE

VIPLE is a visual language for IoT/robotics programming developed at the Arizona State University [58]. Initially developed based on Microsoft Visual Programming Language that was later on discontinued, the VIPLE language continues to support the Microsoft community on visual programming. The language has drawn considerable traction in educational settings and is known for its compatibility with LEGO Mindstorms EV3 robotic programming basics and the underpinning communication technologies (e.g., Wifi, Bluetooth, JSON). The visual editor associated with the VIPLE language permits the declaration of application variables and the pin configuration of the IoT system or robotic application being developed. The source code generated in VIPLE covers non-trivial programming constructs including parallel and event-driven processing. But it sustains limitation regarding the scope of programming variables and as such needs fine-tuning to be useful.

[4]More than 50 proposals of model-to-code transformation from state machine were systematically reviewed.

TABLE 2. State-of-the-art analysis with regard to our research questions: RQ1 – Visual design, RQ2 – Separation of concerns, RQ3 – Code generation.

Framework	RQ1	RQ2	RQ3
Midgar	Almost	Limited	Almost
Asm2C++	Full	Limited	Almost
IoTSuite	Limited	Almost	Limited
UML4IoT	Almost	Limited	Limited
VIPLE	Full	Almost	Limited
EL4IoT	Limited	Limited	Limited
ThingML	Almost	Limited	Limited

6) EL4IoT

EL4IoT [33] is a DSL framework that targets low-end IoT devices and generates software code to be run on Contiki – an open source operating system for low-power IoT devices. This framework promotes the concept of design automation to mitigate the burden of configuring and deploying on Contiki OS. Engineering a DSL for IoT-enabled operating systems is one of the contributions of this work. In addition, EL4IoT facilitates an automatic generation of code for low-end devices in IoT applications that require an IoT-compliant layer to provide interoperability and seamless connectivity to the Internet.

7) ThingML

ThingML is an approach that includes a modeling language and a source code generator for IoT applications [30]. ThingML relies on two key structures to deliver its modeling expressiveness: Things that represent IoT devices and Configurations that describe the interconnection between Things. ThingML generates source code that spawns three contemporary programming languages (C/C++, Java and Javascript). ThingML falls short when it comes to software abstraction and thus does not facilitate the concept of dual-faceted design.

8) COMPARISON

The aforementioned frameworks outline the big picture of MDE-based code generation that supports the development of IoT-based software systems. They exhibit the following major units in common.

- An editor for visual design in the early phase of software/system development
- An underlying model-to-code transformation technique that transforms the high-level specification being edited, coupled with the technical details of IoT components being used into platform-independent executable code
- A deployment module that produces device-specific code, resulting in a distributed architecture that is collaboratively hosted. Technically, such a module relies on a mapper and a linker to perform its job.
- A runtime environment – typically built on top of an existing middleware platform, for the distributed execution of the IoT-based software system in question.

Table 2 shows our analysis on the above-mentioned frameworks with respect to the research questions we have formulated for code generation.

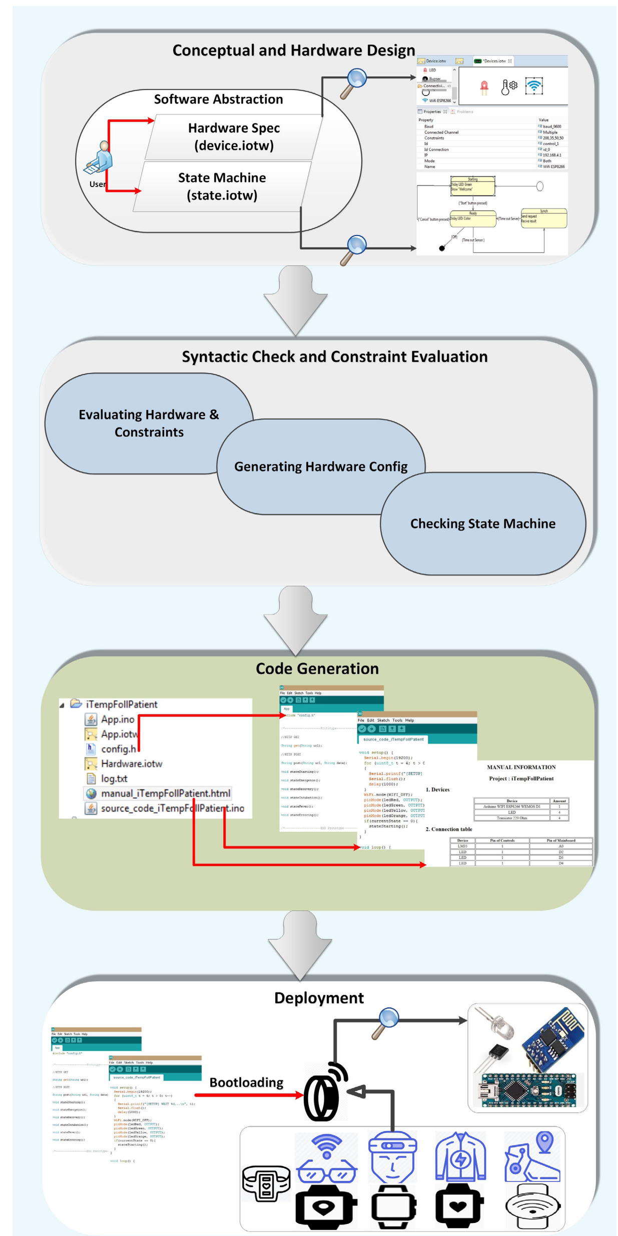


FIGURE 3. Phase-by-phase construction of a wearable device in which the design & development activities are aided with.

IV. MICRASPIS AS A COMPUTER-AIDED TOOL FOR PROGRAMMING IoT WEARABLES

This section highlights how we address our research questions while demonstrating our tool's main features by walking through the previously described case study. Before going into the specific features of our tool, we briefly present a phase-by-phase flow for the construction of wearable devices that the tool might fit into, as shown in Figure 3. This flow, which primarily targets the development and deployment of a wearable, consists of the following four phases[5]: Conceptual & Hardware Design, Syntactic

[5]Due to paper format, we choose to present our flow in a layered style though it might alternatively be presented in a spiral manner to be more eye-pleasing. For this reason, we interchangeably refer to the phases as layers.

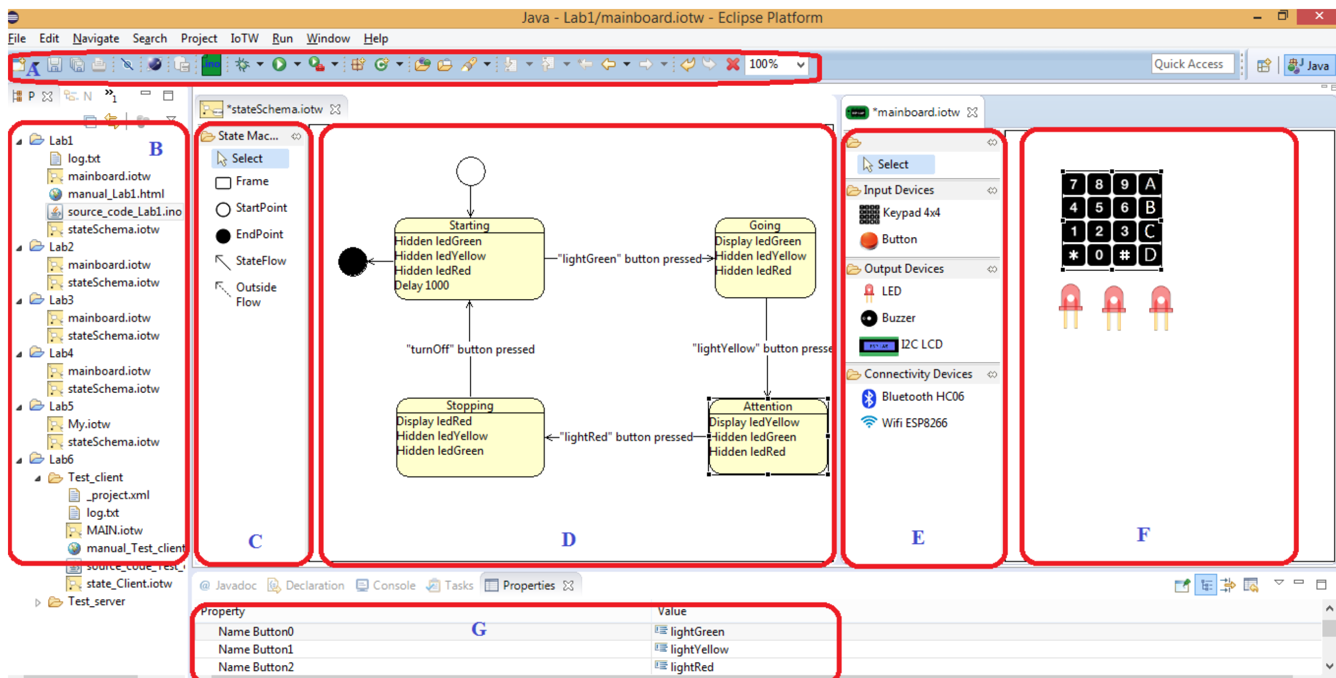


FIGURE 4. **A typical screenshot of Micraspis demonstrating its main widgets and windowing panes that are geared up for effectively working out the multifaceted design of an IoT wearable.**

Check & Constraint Evaluation, Code Generation, and Deployment. Even though Micraspis does not fully span these four phases, it's worth mentioning how the tool assists in developing and deploying an IoT wearable to facilitate the enactment of business processes where the said wearable device plays a role.

- *Conceptual and Hardware Design*. This layer provides a graphics editor that allows developers to visually sketch the hardware architecture of and a behavioral representation for a wearable device. Both of them are subject to being checked against well-formnedness criteria in the next layer.
- *Syntactic Check and Constraint Evaluation*. It is crucial to make sure that the multifaceted design of an IoT wearable is well-formed before any attempt to generate the programming source code could be made. This notion of well-formedness refers to both the user-selected hardware configuration and the logical appropriateness of the wearable's state machine. Micraspis is equipped with a set of hardware constraints and a syntax for text annotation in the state machine that together kick in whenever design-time changes are made to the wearable in question. Once the state machine and the hardware configuration being edited have all passed this well-formedness validation, the highlight is given to the next layer for the sake of code generation.
- *Code Generation*. It is the goal of this layer to produce source code in C++ and auxiliary files to speed up the development and deployment of IoT wearables. We aim to establish a development baseline for programming a wearable device on a specific IoT programming platform (e.g., Arduino). Unlike skeleton code that superficially lists class members and methods without providing a non-trivial implementation of the listed methods, the code generated in this layer should functionally be operational (if not fully operational).
- At the *Deployment* layer, developers fine-tune the generated source code and have it loaded into their IoT wearable to make the device functionally ready for the enactment of business processes it participates in.

We relate the above-mentioned layers to our research questions (see Subsection II-B) as follows: Conceptual and Hardware Design addresses the research question of RQ1; Syntactic Check and Constraint Evaluation targets the question of RQ2; Code Generation – RQ3.

A. CONCEPTUAL AND HARDWARE DESIGN

To get started in constructing a wearable device, we sketch a conceptual design and select hardware components for a component-assembly design. Micraspis features a graphics editor (see Figure 4) just like many other design tools to help developers in this regard.

Figure 4 shows a screenshot of Micraspis demonstrating its most typical windowing areas that are enumerated using capital letters. Area Ⓐ shows a toolbar featuring shortcuts that provide access to frequently-invoked commands. Area Ⓑ depicts a project explorer to be used for navigating through a workspace under Micraspis. Area Ⓒ showcases a palette containing essential elements for describing a state machine graphically. Area Ⓓ illustrates such an editing pane for state machine. Similarly, area Ⓔ showcases another palette containing visual elements for a hardware design which is

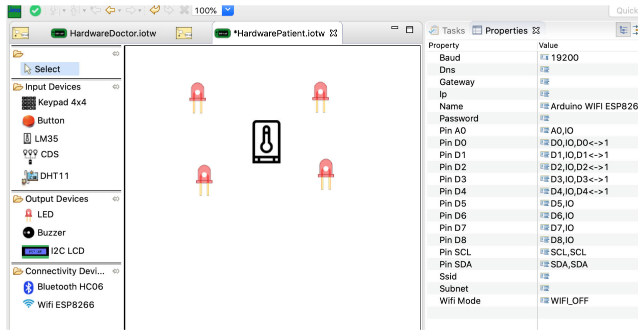


FIGURE 5. **Hardware configuration of the patient's wearable in use for *iTempFoll*.**

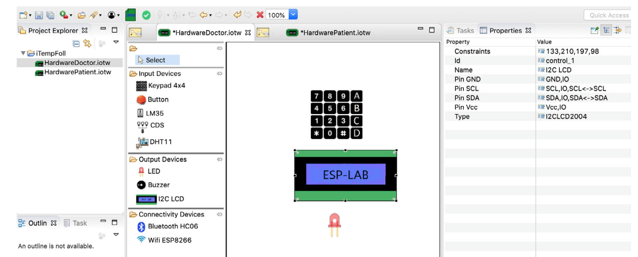


FIGURE 6. **Hardware configuration of the doctor's wearable in use for *iTempFoll*.**

illustrated in area Ⓕ. Lastly, Micraspis provides a property box for viewing and editing the details of a design element as highlighted in area Ⓖ.

Figure 5 and Figure 6 are additional screenshots of Micraspis that demonstrate how we graphically specify the hardware structure of the two wearable devices in use for *iTempFoll*. In terms of hardware design, the patient's device is made up of the following: four light-emitting diodes (LEDs) for displaying the status of the patient using a color range (i.e., red for an emergency, orange – dangerous, yellow – inception, green – recovery, and blue – ready), a temperature sensor, and a Wifi module for communicating with the server. Meanwhile, the doctor's device features the following components: an LCD for briefly displaying the patients' information, a keypad for browsing the patients being display, an LED that is illuminated while the wearer is receiving an alert, and a Wifi module for communicating with the coordinating server.

In Micraspis, a developer may switch between conceptual modeling and hardware design at any time. Both of these views offer a palette and a visual design area into which she could simply drag-and-drop an element from the palette. Micraspis directly supports conceptual modeling in the form of state machines. Figure 7 shows the state machine of the patient's wearable being edited in Micraspis. Each patient's wearable may be in one of the following states at runtime: `Starting`, `Listening`, `Emergency`, `Recovery`, `Incubation`, and `Fever`, the first of which refers to the moment when the wearable is switched on and establishes a Wifi connection to the coordinating server. The wearable makes a transition to `Listening` when ready. In this

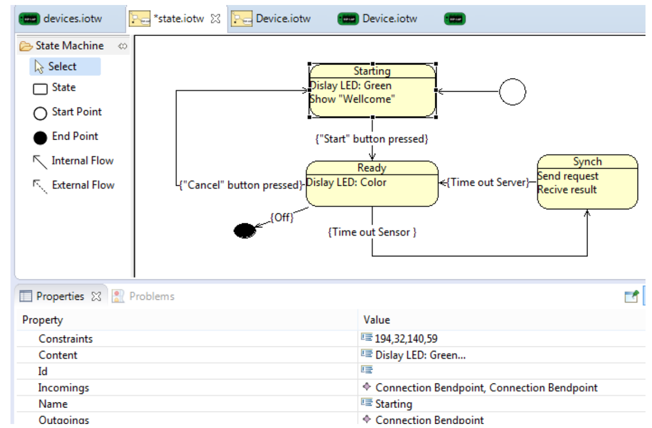


FIGURE 7. **A state machine diagram being edited for the patient's wearable.**

state, the wearable constantly takes a reading of its wearer's body temperature to communicate with the server. Should the server decide to activate a medical rule, this wearable will be asked to change its state to either `Emergency`, `Recovery`, `Incubation`, or `Fever`. The wearable may revert to `Listening` at any time upon receiving a confirmation from the server telling that she has received proper medical treatment.

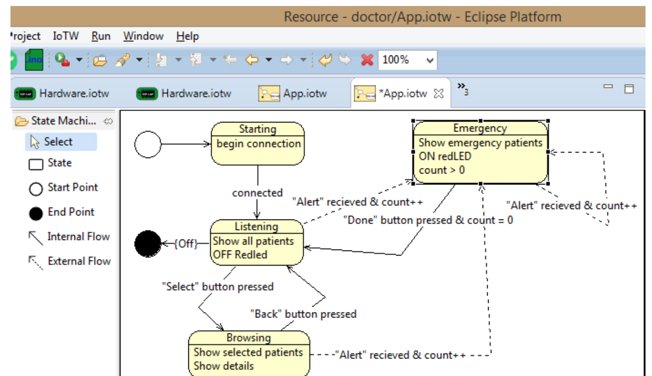


FIGURE 8. **A state machine diagram being edited for the doctor's wearable.**

Similarly, Figure 8 is a screenshot of Micraspis showing the state machine of the doctor's wearable that is being entered for the sake of software abstraction. This device is in `Starting` state when booting it up, followed by `Listening` when the booting is done successfully. At this moment, the wearable displays all patients whom the wearing doctor is assigned to it. The wearable makes a transition to state `Emergency` if it receives an alert (or more) from the coordinating server. At the same time, an alert message pops up in its LCD showing details of the patient who needs to be taken care of. The wearable might receive another alert while its wearer is on her job, resulting in a new record being registered but no change to its state.

The wearing doctor is supposed to hit a designated key in a keypad to notify once she has successfully delivered a

medical action. This wearable may revert to `Listening` state when and only if no more patients are on the waiting list. When in `Listening` state, the wearer may want to view additional information of a patient. Should she take this function, her wearable is switched to `Browsing` state and back to `Listening` state when she exits.

B. SYNTACTIC CHECK AND CONSTRAINT EVALUATION

As a design tool, Micraspis comes with a set of constraints and syntactic rules. We may not proceed with code generation without ensuring that both the hardware design and behavioral specification of a wearable device being constructed are well-formed. To this end, Micrapis undergoes two sub-phases that both run in background to ensure this well-formedness. The first sub-phase aims at the rationality of the device's architecture by iterating through a list of predefined hardware constraints. The second sub-phase is dedicated to the syntactic check that scans all text annotated to the state machine being edited to check if there are any irregularities.

1) SUB-PHASE 1: CONSTRAINT EVALUATION

Speaking of the hardware architecture, the wearable has a mainboard that harbors hardware components. Micraspis arranges a configuration of the mainboard pins through which these components could be attached.

As described in Table 3, Micraspis reinforces the following hardware constraints: (i) no more than one keypad/LCD/Wifi; (ii) there are enough mainboard pins of each kind for all hardware components; (iii) pin arrangement between hardware items and the mainboard is attainable. Any time an addition or modification is made to this hardware design, the tool (re-)evaluates the above-mentioned constraints and (re-)configures the pins to match the type of each hardware components if needed. Should one of the above constraints be violated, the tool issues a warning immediately.

TABLE 3. List of hardware constraints in Micraspis.

Constraint	Meaning
Maximum number of pins	The number of digital I/O pins on an R3 Arduino board is 14 and 6 Analog Input Pins
Maximum number of input pins	The maximum number of digital input pins on an R3 Arduino board is 12
Maximum number of output pins	No more than 8 digital output pins on an R3 Arduino board
Lone keypad	The maximum number of keypads plugged into an R3 Arduino board is one
Lone LCD	No more than one LCD should be plugged into an R3 Arduino board
Lone Wifi	No more than one Wifi should be plugged into an R3 Arduino board
Lone Bluetooth	No more than one Bluetooth should be plugged into an R3 Arduino board
Input components	At least one input component plugged into the mainboard
Output components	At least one output component plugged into the mainboard

TABLE 4. Textual annotations in a state machine edited in Micraspis should follow a syntax.

Syntax	Description	Components	Example
`Display` <component>	Turn on a component	LED	Display redLED
`Halt` <component>	Stop a component	LED, I2LCD	Halt redLED
`Blink` <compoment>	Blink a component	LED, I2CLCD	Blink redLED, Blink myLCD
`Show` <String>	Display a string on a component	I2CLCD	Show "Welcome"
`Beep` <id>	Sound of a component	Buzzer	Beep myBuzzer
<String> button `pressed`	Button is pressed	Keypad	"Cancel" button pressed
<id> `pushed`	Receive a push button event	Button	Arlambutton pushed
<String> `sent`	Send data via Wifi	Esp8266 Wifi	bodyTemp sent
<String> `received`	Receive data via Wifi	Esp8266 Wifi	bodyTemp received

2) SUB-PHASE 2: SYNTACTIC CHECK

As mentioned, Micraspis allows developers to specify the behavior of wearable using a state machine (as exemplified in Figure 7, Figure 8). In this sub-phase, operations declared for each state and text annotated to transitions in the state machine are validated against a predefined grammar (see Table 4). As we typically make reference to the selected hardware items in this state machine via their identifiers, this syntactic check should be followed by the constraint evaluation previously performed on the hardware design in Micraspis.

C. CODE GENERATION

Micraspis is available as a plugin of Eclipse – a widely used integrated development environment in computer programming. We rely on a meta-modeling framework in Eclipse, called Eclipse Modeling Framework [59], to facilitate the code generation (see Subsection IV-C2 and Subsection IV-C3). In Micraspis, we enjoy code generation for the following programming aspects of an IoT wearable: (i) generating a switch-case statement to implement the main control loop; (ii) factorizing functions for most state transitions and state operations; (iii) automatically arranging pins to which hardware components are wired up to the mainboard; (iv) producing technical documentation.

Code generation from a state machine is by far more difficult to accomplish than otherwise from a class diagram [52], due primarily to the lack of direct and precise mapping between concepts borrowed from state diagrams and general-purpose programming languages. For example, there is no counterpart of a transition or a state in contemporary object-oriented programming languages. Unlike the so-called skeleton code that could straightforwardly be produced out of a class diagram, elements of a UML statechart diagram such

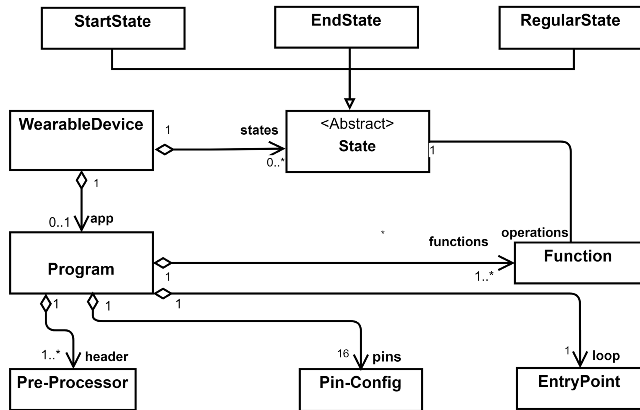


FIGURE 9. Meta-model of a wearable's computer program in Micraspis.

as state, event, transition exhibit behavioral semantics that should be translated into the implementation of functions and methods in general-purpose programming. The underlying code generation of Micraspis kicks off when the syntactic check is done for these elements.

1) AN UNDERLYING CODE GENERATION ENGINE

Wearable devices typically function in an asynchronous mode. Its main control loop is there to listen to state-changing events. Like the operations of a state, the events could be factorized as functions. Our tool takes a state machine as the input for generating the implementation of all these functions. For example, `OFF greenLED` in a statechart diagram will translate into `digitalWrite(greenLED, LOW);` when programmed in C++. Note that identifiers (such as `greenLED`) given to hardware items being selected in the hardware design are put as programming variables when generating code. A self-transition (e.g., `Emergency` has a transition that returns to itself) will boil down to a recursive function in C++.

The underlying code generation engine of Micraspis programmatically allocates mainboard pins to components and composes a document detailing this pin configuration according to which the chosen components will be wired up physically. This could be kept as a reference for wearable developers (for instance, Uno Arduino (R3) consists of 14-digital I/O pins, 6 of which can be used as pulse width modulation outputs). While generating the code, the engine keeps track of external libraries that are going to be imported when preparing the programming environment for the generated source code. To this end, apart from generating source code in C++ that turn the state machine and hardware design of the wearable device being constructed into a functional computer program, Micraspis additionally produces the following two files: (a) an HTML file explaining the pin layout that instructs the developer how to wire up the selected hardware components to the wearable's mainboard; (b) an XML file that describes external libraries needed to successfully compile the generated C++ code.

2) ABSTRACT SYNTAX

Let us express the abstract syntax behind Micraspis's code generation engine using a meta-model as shown in Figure 9 and Figure 10, which correspond to the behavioral specification and the hardware design of the wearable device being constructed, respectively. In Figure 9, the concept in focus would be *State* of which the subclasses represent the start/end state and regular states (concept *RegularState*). In line with the UML statechart diagram, there should be a single start state, one or more end state(s), and of course multiple regular states in a state machine being edited. However, we would not allow nested states in Micraspis, expecting the state machine to be represented in a flat structure. As indicated in the meta-model, all states (the start and end ones included) may have operations specified within. A computer program that operates a wearable device (concept *Program*) is programmatically composed of several functions in C++. Additionally, it includes necessary pre-processor statements, an entry point, and the definition of constants needed for configuring the wearable's pin layout. Note that both the operations that are visible in the state machine and the functions in C++ code that will be generated by our tool convergingly point to concept *Function* in our abstract syntax.

Figure 10 is an excerpt of our meta-model capturing all hardware pieces available to the developer in Micraspis. Concept *Component* is an abstract class that represents all components, which are subclassed into *InputComponent*, *OutputComponent*, and *Connectivity*. Concept *InputComponent* stands for components that offer an input effect to construct a wearable device, including keypads, buttons, and sensors. *OutputComponent* is another subclass that represents hardware pieces producing an output effect (e.g, LED, LCD, buzzer). The *Connectivity* concept facilitate IoT connectivity technologies (e.g., bluetooth, Wifi). As for the mainboard, a wearable device architecturally consists of multiple hardware components and a mainboard of which Arduino boards like *R3Arduino* and *ESP8266Arduino* are currently supported in Micraspis. The mainboard harbors a number of pins (concept *Pin*) through which the hardware components of a wearable device may be wired up (hence, a UML association between *Component* and *Pin* in this abstract syntax).

3) CONCRETE SYNTAX

The concrete syntax for what is presented in Figure 9 is basically UML-like. Directed arrows denote state transition, and a rounded rectangle signifies a regular state. Operations are visible in its lower compartment, and we allow operations to be specified for any kind of state in Micraspis. They are diagrammatically represented for regular states. For start/end states, while they may not be represented visually, they are still listed in an editable fashion in a property box if a start/end state is selected in the graphics area. The concrete syntax for hardware components (for which the abstract syntax is depicted by Figure 10) is rather straightforward, where icons are extensively used to show them visually. In line with our

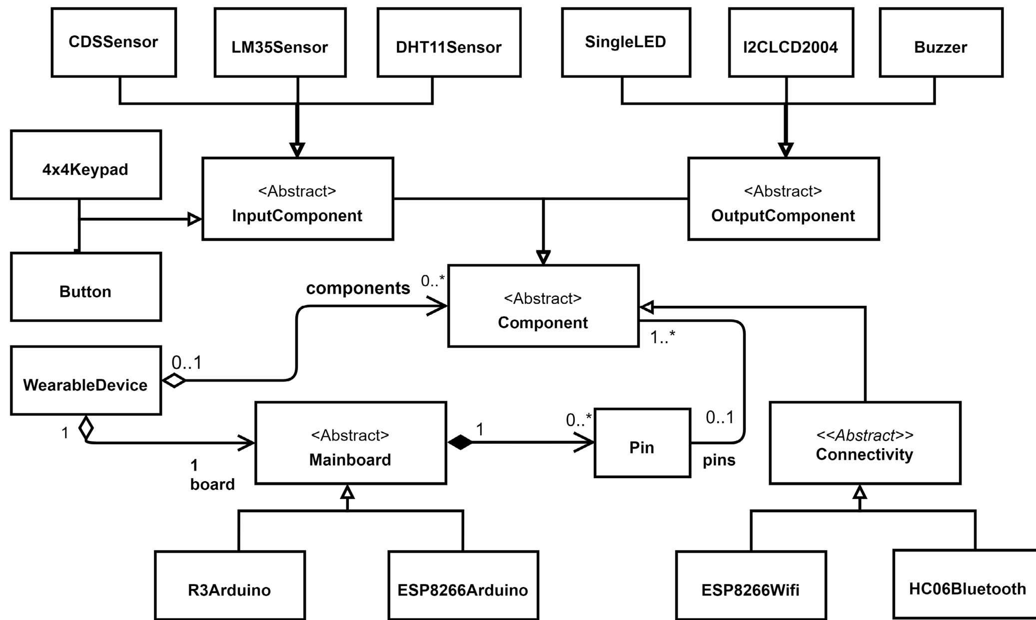


FIGURE 10. Meta-model of hardware components that are currently supported by Micraspis.

abstract syntax, these icons are not supposed to be connected using lines or arrows in the editing pane of Micraspis. Moreover, these hardware pieces are not hierarchically organized, meaning their concrete syntax is of a flat structure – as can be seen in the illustrated screenshots of our tool.

V. VALIDATION

Micraspis tool has been tested in a series of mini lab projects with a logical progression to complexity. In this section, we present them in detail (Subsection V-A) and report the feedback we obtained from those who worked with our tool in this series of lab projects (Subsection V-B).

A. MINI LAB PROJECTS

The mini lab projects were designed to incrementally experiment on how Micraspis assists developers in assembling and programming wearable device(s). In Table 5, we report the mini projects[6] as follows (from left to right): project's short name, total number of lines of code generated thanks to Micraspis, total number of lines of code when fully implemented, hardware components needed to assemble the wearable(s) in the said project, how many states that were conceptually represented. The rightmost column of Table 5 tells us the degree of code completion in the said project as a metric for measuring the effectiveness of having source code produced by our tool. Note that these mini projects are placed in an order with a logical progression to complexity. The first one in this list, which is a mere programming exercise that simulates traffic lights, intentionally designed for novices. The last project in this list is the most non-trivial one, detailing the construction of the doctor's wearable and patient's wearable as featured in our *iTempFoll* case study.

[6]Full description could be found at https://tinyurl.com/y8yuqu6v

TABLE 5. Computer-aided programming in mini projects thanks to Micraspis.

Project	LOCs aided	LOCs done	Components	N° of States & N° of Tasks	% Code Done
Traffic Light	58	58	LED, keypad	4 states & 12 tasks	100%
LED Passing	70	90	LED, Button	3 states & 6 tasks	78%
Welcoming Screen	82	99	LCD, Keypad	2 states & 2 tasks	83%
Alarming Devices	102	142	LED, Buzzer, Keypad	3 states & 7 tasks	72%
Wifi Signaling Device	148	197	LED, Wifi	3 states & 8 tasks	75%
iTempFoll	172	265	LED, Wifi, LCD, Keypad, Sensor	7 states & 12 tasks	65%

B. FEEDBACK FROM PRACTITIONERS AND STUDENTS

To see how Micraspis addresses the three challenges of offering a computer-aided solution to the construction of wearable computers (see Subsection II-B), we invited practitioners in IoT-related domains and bachelor students having fundamental understanding of IoT-enabled smart systems to validate Micraspis and have their say.

1) PROTOCOL FOR OBTAINING USERS' FEEDBACK

A questionnaire was prepared to obtain feedback on Micraspis from practitioners whose background was in

IoT-related domains. Questions asked are largely about the extent to which Micraspis addresses the three challenges that are presented in Section II-B. To help them get acquainted with Micraspis before actually filling out this questionnaire, we provided them with a tutorial detailing step-by-step instructions for working with the Micraspis tool. The tutorial and the questionnaire were tested within our research group to estimate the time needed for a participant to complete the validation and to remove ambiguity if any. A total of 31 software engineers from 16 software development companies took part in our validation. They reportedly had 2-15 years of experience in software development and computer engineering.

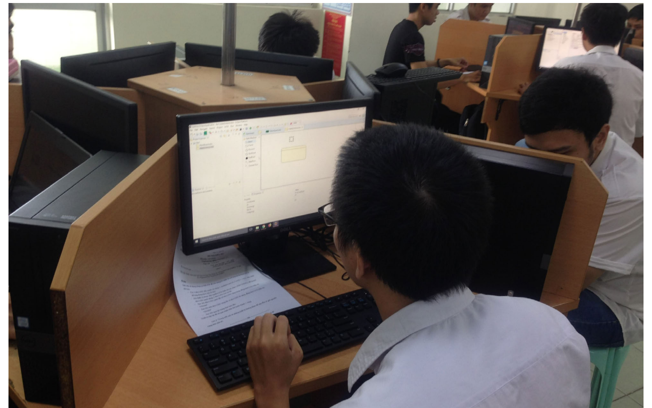
Undergraduate students were asked to undertake the pre-defined mini lab projects (see Subsection V-A) on four university campuses in Ho Chi Minh City, namely University of Technology, Industrial University, University of Information Technology, and Cao Thang College. Figure 11 gives an example of a group of undergraduate students working with our tool to get their programming tasks done in one of the mini lab projects.

Due to the difference in the software practitioners' availability, we decided to proceed with them individually. Where face-to-face conversation was possible, the participants followed the tutorial and then filled out the questionnaire themselves while receiving assistance either on the phone or via an online chat. For those students who were willing to take part in the validation, we organized them into groups of three to five and guiding them more thoroughly. Instead of following a pre-built scenario, the students performed extensive tool-based practice on Micraspis through constructing a wearable themselves. In total, we invited 161 practitioners/students to participate in this validation.

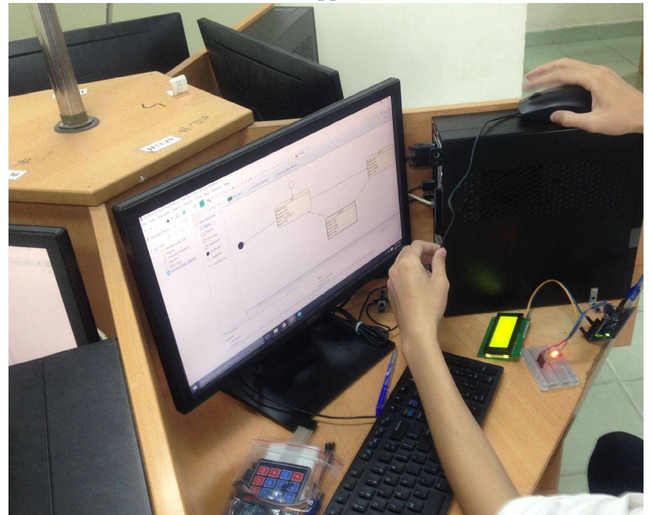
2) RATINGS

A total of 130 students took part in the validation of Micraspis. Most of them were junior whose major was in computer science. A few of them were sophomore that all had taken fundamental programming courses. They participated in this validation while taking a bachelor course on IoT programming. In this course, they were divided into groups of less than five. Each group represented a course project with an aim to make a fully operational IoT wearable. They then answered a questionnaire to assess how Micraspis addressed the three tool requirements and to give their feedback as if Micraspis was employed as a teaching tool in their course.

We collected feedback of 31 practitioners, making 161 the total number of respondents. Table 7 presents the breakdown of their answers to the questions asked (being visualized in Figure 12). All questions but the last were designed with the Likert scale. Remarkably, the respondents overwhelmingly agree that Micraspis should find more applicability beyond health informatics in today's IoT-enabled smart society (as depicted by answers given to Q7 in the questionnaire). Q2 and Q5 in the questionnaire are where Micraspis performs noticeably well, suggesting that the programmers generally are happy with the amount of code being generated and appreciate the pin layout being arranged automatically by Micraspis.



(a) Working with Micraspis to generate source code to shorten the time to market of IoT-enabled applications.



(b) Checking C++ source code that was generated thanks to Micraspis before bootloading.

FIGURE 11. A group of undergraduate students working with the micraspis tool on the campus of industrial university of Ho Chi Minh city.

On the downside, hardware components available to programmers in Micraspis are not plentiful, as evidenced by the answers we received for Q1 from the questionnaire. This question is where the highest level of disagreement is recorded. The respondents were not particularly impressed by model-driven engineering techniques (and state machine modeling in particular) being offered by the tool, as indicated by the relatively low satisfaction rate for Q3, Q4 and Q6 from the questionnaire. Interestingly, questions Q1, Q3, Q4, and Q6 draw a mixture of respondents' agreement and disagreement.

Despite some negative feedback recorded, the answers we obtained have not statistically rejected any research questions we have identified. As can be seen in the rightmost column of Table 6, we associate each interviewing question of the questionnaire with a particular research challenge we have formulated in Section II. If we filter out the feedback left for

TABLE 6. List of interviewing questions.

Question	Description	Linked to
Q1	Are hardware components available in Micraspis all components that you need for developing IoT wearable applications?	*RQ1*, *RQ3*
Q2	Are you, as a developer, fully freed from programmatically configuring the pins of your hardware components and the mainboard of your wearable?	*RQ1*, *RQ3*
Q3	To what extent does Micraspis address the semantic links between your hardware specification and your application logic (i.e., UML state machine)?	*RQ2*
Q4	How much reuse do you enjoy when moving from an IoT platform to another in programming your wearable applications thanks to Micraspis?	*RQ2*
Q5	To what extent does source code generated by Micraspis cover for the implementation of your wearable application?	*RQ1*, *RQ3*
Q6	Does Micraspis offer you enough visual model-based techniques for describing your IoT wearable application?	*RQ1*, *RQ3*
Q7	In your opinion, Micraspis directly supports wearable development in the following three domains: smart home, smart hospital and smart restaurant?	*All*
Q8	Other remarks:	*All*

TABLE 7. Breakdown of answers given by those who were interviewed on the usability and capability of Micraspis, 19.3 % of whom were engineers having experience in IoT domains.

Question	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Strongly Disagree	3	0	1	1	1	1	0
Disagree	21	6	5	10	12	10	10
Neutral	31	34	39	35	28	45	27
Agree	*82*	*63*	*75*	*68*	*67*	*62*	*58*
Strongly Agree	*24*	*58*	*41*	*47*	*53*	*43*	*66*
Satisfaction Rate %	**65.8**	**75.2**	**72.0**	**71.4**	**74.5**	**65.2**	**77.0**

all interviewing questions that are associated with a given, previously stated research challenge, we will not see that Micraspis markedly disappoints the programmers for any of these research challenges. Those respondents who were not happy with Micraspis might have found that the presentation of the prototype was a little unappealing and cumbersome as they have to click on many items before finally working out their way to accomplish code generation.

C. LESSONS LEARNED: LACK OF PLAUSIBLE WAYS TO MEASURE TIME SAVING

A natural question that readers might ask is to how much our computer-aided proposal helps shorten the time to market of wearable devices. As we have described a metric for measuring the productivity of code generation in Table 5, our attention is now turned into reporting the time saving thanks to Micraspis when developing the wearables mentioned in our lab projects. Let us explain why we literally had no plausible ways to measure this time saving objectively.

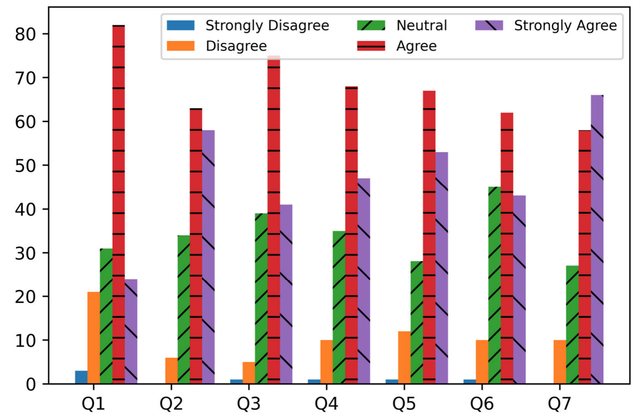


FIGURE 12. A bar chart that visualizes answers we collected for our questionnaire about Micraspis.

For each mini lab project, we wished to conduct two experimental development cycles of which the output (i.e., having the said wearable(s) fully operational) and the human resource (i.e. an individual or a team who carried out the experiments) are identical. Purposefully, one of them was with and the other without an aid of Micraspis. However, simply recording the time diffidence of these two development cycles may not precisely reflect the time saving attributed to Micraspis, as there were many unaccountable factors. For example, our respondents could have already gotten acquainted with the structure of the source code and the component-assembly design of the wearable(s) in question before heading for the latter experimental development cycle, making the total time and effort significantly lower compared to those in an alternative scenario where the latter experiment otherwise went first. Furthermore, we were unable to make sure that our respondents paid the same amount of attention to programming and hardware design when participating in these dual experiments on the same mini project.

VI. FINAL CONSIDERATIONS

The future of IoT is shaped by wearable technology – this is not just a common saying. Wearable computing enables new business models and software architecture that might otherwise be impossible [3], [45]. While a coordinating server is expected to perform decision-making and other non-trivial tasks in a typical IoT-based architecture, it relies on small body-worn wearable devices for collecting measurement data from people wearing them and dispatching instructions. Wearable devices arguably play a key role in this ever more distributed architecture, giving rise to IoWT as a sibling term of IoT. Enabling hardware components and wearable technologies continue to advance, yet effectively programming and deploying them poses a few challenges. First, wearables are assembled and programmed to for their wearer to participate in certain business processes, of which the enactment can be tweaked at any time due to business changes. Furthermore, a computer program already deployed on

a wearable device is subject to software rebuild to keep pace with technological changes (e.g., newly available hardware components). Moreover, there is is still a programming burden involved with data marshalling in IoWT programming, which by nature is error-prone. Generating source code to ease this painstaking task, though might sound an old school way of software automation, is worth revisiting. While research frameworks that target IoT programming look abundant (e.g., IoTSuite [40], Midgar [55], Asm2C++ [57], Viple [58], ThingML [30]), computer-aided proposals to IoWT programming are anything near non-existent. Our standpoint in this realm is to leverage model-driven engineering techniques, in particular model-to-code transformation approaches that are tailored for the state machine. We identify the following three requirements for a tool that supports wearable programming: (i) dual-faceted visual design; (ii) separation of concerns that enable model transformation; (iii) dedicated code generation techniques. Our research has resulted in a computer-aided tool called Micraspis that allows a wearable to be described both behaviorally and architecturally. The tool produces source code in C language featuring an entry point, regular functions that respond to events captured in the said state machine, and necessary variables that represent the pin layout of all hardware components.

We conducted a couple of experiments to measure the effectiveness of our tool proposal. On the one hand, we measured the amount of source code generated relative to what is needed for having a wearable device fully operational. On the other hand, we obtained feedback from programmers on how they rated Micraspis in meeting the research challenges we formulated. Measurably, code generation done by Micraspis amounts to at least 60% of the full implementation in a series of 6 mini IoWT projects. Most of the programmers we interviewed express their positive attitude to how Micraspis addresses the research questions we formulated for IoWT programming. In total, 161 programmers responded to a Likert scale questionnaire, with which almost 66% of the respondents either agreed or strongly agreed.

Our plan is to release a Web-based version of Micraspis. Work is currently underway to enrich the underlying model-to-code templates to make code generation possible for a couple of variants of the target platform for IoWT programming, e.g., Due Arduino, Mega Arduino, Raspberry Pi. It is known that a state machine might be represented at several levels of granularity. We make this choice available to the wearable programmers in the future versions of Micraspis for the sake of design flexibility. We will reflect the programmers' opinion we collected from the questionnaire in launching the next substantially enhanced version of our tool (e.g., more hardware components and additional business domains for IoWT programming).

Discussions: Architecting and programming wearable devices is part of the business process enactment for IoT-enabled smart solutions. To make the enactment successful, the coordinating servers and workstations that perform non-trivial computing tasks behind the scene (see our *iTempFoll* case study) need to be programmed in concert. The current version of our tool has nothing to do with the design and implementation of software modules deployed on these servers. Methodically, in order to fully support the enactment of IoT-enabled processes, our computer-aided proposal should cover this server-side programming, which in fact might be done in a language that is different from the target language of IoWT programming. For a relatively complex IoT wearable device (e.g., having a great number of states, each of which comes with a lot of operations), it is crucial to prove that our model-to-code transformation is theoretically sound. To achieve this rigor, we aim to follow the principle of verified model-to-code transformation [60] in the future releases of Micraspis. It should be pointed out that the operations and transitions of the state machine being edited in Micraspis are wired to the programming functions in the target source code. While this approach makes code generation straightforward and verifiable, it sounds too rigid and might not be welcome by business analysts – who rather prefer a lift in the abstraction level. The development of a wearable device may be pat of a multidisciplinary system engineering project where non-computing sciences and engineering do matter [45]. We wish to make it explicit that our tool Micraspis has nothing to do with non-computing aspects of the craftsmanship of a wearable device, e.g., material science & engineering, human factors & ergonomics.

ACKNOWLEDGMENT

The authors acknowledge the support of time and facilities from the Ho Chi Minh City University of Technology (VNU-HCM) for this study. They would like to express their sincere gratitude to a colleague of Hoang-Anh Pham for his countless constructive comments on the hardware design facet that has been materialized in Micraspis, especially in its early-phase prototypical development.

REFERENCES

[1] Mordor Intelligence. (2020). *IoT Chip Market–Growth, Trends, COVID-19 Impact, and Forecasts (2021–2026)*. [Online]. Available: https://www.mordorintelligence.com/industry-reports/iot-chip-market

[2] A. Ometov, O. Chukhno, N. Chukhno, J. Nurmi, and E. S. Lohan, "When wearable technology meets computing in future networks: A road ahead," in *Proc. 18th ACM Int. Conf. Comput. Frontiers*. Rome, Italy: ACM, May 2021, pp. 185–190.

[3] S.-E. D. Silva, *Examining Developments and Applications of Wearable Devices in Modern Society*, R.-A. Oliveira and A.-A. Loureiro, Eds. Hershey, PA, USA: IGI Global, Aug. 2017.

[4] C. Fiandrino, N. Allio, D. Kliazovich, P. Giaccone, and P. Bouvry, "Profiling performance of application partitioning for wearable devices in mobile cloud and fog computing," *IEEE Access*, vol. 7, pp. 12156–12166, 2019.

[5] J. Rodrigues, D. Segundo, H. Junqueira, M. Sabino, R. Prince, J. Al-Muhtadi, and V. Albuquerque, "Enabling technologies for the internet of health things," *IEEE Access*, vol. 6, pp. 13129–13141, 2018.

[6] M. M. Alam, H. Malik, M. I. Khan, T. Pardy, A. Kuusik, and Y. L. Moullec, "A survey on the roles of communication technologies in IoT-based personalized healthcare applications," *IEEE Access*, vol. 6, pp. 36611–36631, 2018.

[7] B. Elsa Baby, S.-B. Kadam, and A. Aravindhan, "Human health monitoring system using internet of wearable things," in *Emerging Technologies for Sustainability*, 1st ed., P.-C. Thomas, V.-J. Mathai, and G. Titus, Eds. New York, NY, USA: Taylor & Francis, 2020, pp. 253–259.

[8] S. Khan and M. Alam, "Wearable Internet of Things for personalized healthcare: Study of trends and latent research," in *Health Informatics: A Computational Perspective in Healthcare*, R. Patgiri, A. Biswas, and P. Roy, Eds. New York, NY, USA: McGraw-Hill, 2021, pp. 43–60.

[9] K. A. Eldrandaly, M. Abdel-Basset, and L. A. Shawky, "Internet of spatial things: A new reference model with insight analysis," *IEEE Access*, vol. 7, pp. 19653–19669, 2019.

[10] N. Zuidhof, S. Ben, O. Peters, and P.-P. Verbeek, "A theoretical framework to study long-term use of smart eyewear," in *Proc. Adjunct ACM Int. Joint Conf. Pervas. Ubiquitous Comput., ACM Int. Symp. Wearable Comput.*, Sep. 2019, pp. 667–670.

[11] W. B. Qaim, A. Ometov, A. Molinaro, I. Lener, C. Campolo, E. S. Lohan, and J. Nurmi, "Towards energy efficiency in the internet of wearable things: A systematic review," *IEEE Access*, vol. 8, pp. 175412–175435, 2020.

[12] F. J. Dian, R. Vahidnia, and A. Rahmati, "Wearables and the Internet of Things (IoT), applications, opportunities, and challenges: A survey," *IEEE Access*, vol. 8, pp. 69200–69211, 2020.

[13] C. Timothy, *Precision: Principles, Practices and Solutions for the Internet of Things*. New York, NY, USA: McGraw-Hill, 2016.

[14] K. Robson, J. Kietzmann, and L. Pitt, "APC forum: Extending business values through wearables," *MIS Quart. Executive*, vol. 15, no. 2, pp. 167–177, Jan. 2016.

[15] S. Schönig, A. P. Aires, A. Ermer, and S. Jablonski, "Workflow support in wearable production information systems," in *Proc. 30th Int. Conf. Adv. Inf. Syst. Eng. (CAiSE Forum)*. Tallinn, Estonia: Springer, Jun. 2018, pp. 235–243.

[16] B. Morin, N. Harrand, and F. Fleurey, "Model-based software engineering to tame the IoT jungle," *IEEE Softw.*, vol. 34, no. 1, pp. 30–36, Jan. 2017.

[17] K. Thramboulidis and F. Christoulakis, "UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems," *Comput. Ind.*, vol. 82, pp. 259–272, Oct. 2016.

[18] M. Ekstedt, P. Johnson, and I. Jacobson, "Where's the theory for software engineering?" *IEEE Software*, vol. 29, no. 5, pp. 94–95, Sep. 2012.

[19] M. Fowler, *Domain-Specific Languages*, 1st ed. London, U.K.: Pearson, 2010.

[20] L. Jóźwiak, "Advanced mobile and wearable systems," *Microprocessors Microsyst.*, vol. 50, pp. 202–221, May 2017.

[21] M. Iglesias-Urkia, A. Gómez, D. Casado-Mansilla, and A. Urbieta, "Automatic generation of web of things servients using thing descriptions," *Pers. Ubiquitous Comput.*, pp. 1–17, Jul. 2020.

[22] D. Arellanes and K.-K. Lau, "Algebraic service composition for user-centric IoT applications," in *Proc. 3rd Int. Conf. Internet Things*. Seattle, WA, USA: Springer, Jun. 2018, pp. 56–69.

[23] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, and J. Kim, "FACT: Functionality-centric access control system for IoT programming frameworks," in *Proc. 22nd Symp. Access Control Models Technol.* Indianapolis, IN, USA: ACM, Jun. 2017, pp. 43–54.

[24] L. Riliskis, J. Hong, and P. Levis, "Ravel: Programming IoT applications as distributed models, views, and controllers," in *Proc. Int. Workshop Internet Things Towards Appl.* Seoul, South Korea: ACM, Nov. 2015, pp. 1–6.

[25] R. Calegari, E. Denti, S. Mariani, and A. Omicini, "Logic programming as a service (LPaaS): Intelligence for the IoT," in *Proc. IEEE 14th Int. Conf. Netw., Sens. Control (ICNSC)*. Calabria, Italy: IEEE, May 2017, pp. 72–77.

[26] B. Negash, T. Westerlund, A. M. Rahmani, P. Liljeberg, and H. Tenhunen, "DoS-IL: A domain specific Internet of Things language for resource constrained devices," *Procedia Comput. Sci.*, vol. 109, pp. 416–423, May 2017.

[27] T. Mahapatra, I. Gerostathopoulos, C. Prehofer, and S. G. Gore, "Graphical spark programming in IoT mashup tools," in *Proc. 5th Int. Conf. Internet Things, Syst., Manage. Secur.* Valencia, Spain: IEEE, Oct. 2018, pp. 163–170.

[28] A. S. Thuluva, A. Bröring, G. P. Medagoda, H. Don, D. Anicic, and J. Seeger, "Recipes for IoT applications," in *Proc. 7th Int. Conf. Internet Things*. Linz, Austria: ACM, Oct. 2017, pp. 1–8.

[29] G. De Luca, Z. Li, S. Mian, and Y. Chen, "Visual programming language environment for different IoT and robotics platforms in computer science education," *CAAI Trans. Intell. Technol.*, vol. 3, no. 2, pp. 119–130, Jun. 2018.

[30] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: A language and code generation framework for heterogeneous targets," in *Proc. ACM/IEEE 19th Int. Conf. Model Driven Eng. Lang. Syst.* Saint-Malo, France: ACM, Oct. 2016, pp. 125–135.

[31] I. Berrouyne, M. Adda, J.-M. Mottu, J.-C. Royer, and M. Tisi, "CyprIoT: Framework for modelling and controlling network-based IoT applications," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.* Limassol, Cyprus: ACM, Apr. 2019, pp. 832–841.

[32] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione, and R. Spalazzese, "Model-driven engineering for mission-critical IoT systems," *IEEE Softw.*, vol. 34, no. 1, pp. 46–53, Jan. 2017.

[33] T. Gomes, P. Lopes, J. Alves, P. Mestre, J. Cabral, J. L. Monteiro, and A. Tavares, "A modeling domain-specific language for IoT-enabled operating systems," in *Proc. 43rd Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*. Beijing, China: IEEE, Oct. 2017, pp. 3945–3950.

[34] J. Laassiri and S.-D. Krit, "Internet of Things—Architecture and concepts in ODP information language," in *Proc. Int. Conf. Eng. MIS (ICEMIS)*. Agadir, Morocco: IEEE, Sep. 2016, pp. 1–5.

[35] D. A. Robles-Ramirez, P. J. Escamilla-Ambrosio, and T. Tryfonas, "IoT-sec: UML extension for Internet of Things systems security modelling," in *Proc. 8th Int. Conf. Mechatronics, Electron. Automot. Eng. (ICMEAE)*. Bangkok, Thailand: IEEE, Nov. 2017.

[36] T. Eterovic, E. Kaljic, D. Donko, A. Salihbegovic, and S. Ribic, "An Internet of Things visual domain specific modeling language based on UML," in *Proc. 25th Int. Conf. Inf., Commun. Autom. Technol. (ICAT)*. Sarajevo, Bosnia Herzegovina: IEEE, Oct. 2015, pp. 1–5.

[37] C. G. García, B. C. P. G-Bustelo, J. P. Espada, and G. Cueva-Fernandez, "Midgar: Generation of heterogeneous objects interconnecting applications. A domain specific language proposal for Internet of Things scenarios," *Comput. Netw.*, vol. 64, pp. 143–158, May 2014.

[38] C. Steinmetz, G. Schroeder, A. dos Santos Roque, C. E. Pereira, C. Wagner, P. Saalmann, and B. Hellingrath, "Ontology-driven IoT code generation for FIWARE," in *Proc. IEEE 15th Int. Conf. Ind. Informat. (INDIN)*. Emden, Germany: IEEE, Jul. 2017, pp. 38–43.

[39] S. Bonfanti, M. Carissoni, A. Gargantini, and A. Mashkoor, "*Asm2C++*: A tool for code generation from abstract state machines to Arduino," in *Proc. 9th Int. NASA Formal Methods Symp.* Moffett Field, CA, USA: Springer, May 2017, pp. 295–301.

[40] S. Chauhan, P. Patel, A. Sureka, F.-C. Delicato, and S. Chaudhary, "IoT-Suite: A framework to design, implement, and deploy IoT applications: Demonstration abstract," in *Proc. 15th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw.*, Vienna, Austria, Apr. 2016, pp. 37:1–37:2.

[41] M. Sharaf, M. Abughazala, and H. Muccini, "Arduino realization of CAPS IoT architecture descriptions," in *Proc. 12th Eur. Conf. Softw. Archit., Companion*. Madrid, Spain: ACM, Sep. 2018, pp. 1–4.

[42] M. Haghi, K. Thurow, and R. Stoll, "Wearable devices in medical Internet of Things: Scientific research and commercially available devices," *Healthcare Inf. Res.*, vol. 23, no. 1, pp. 4–15, Jan. 2017.

[43] R.-M. Aileni, G. Suciu, M. Rajagopal, S. Pasca, and C.-A. Sukuyama, "Data privacy and security for IoMWT (internet of medical wearable things) cloud," in *IoT and ICT for Healthcare Applications*, N. Gupta and S. Paiva, Eds. New York, NY, USA: McGraw-Hill, 2020, pp. 191–215.

[44] S.-V. Hiremath, G. Yang, and K. Mankodiya, "Wearable Internet of Things: Concept, architectural components and promises for person-centered healthcare," in *Proc. 4th Int. Conf. Wireless Mobile Commun. Healthcare–Transforming Healthcare Through Innovations Mobile Wireless Technol.* Athens, Greece: IEEE, Nov. 2014, pp. 304–307.

[45] R. Benbunan-Fich, "An affordance lens for wearable information systems," *Eur. J. Inf. Syst.*, vol. 28, no. 3, pp. 256–271, May 2019.

[46] L.-P. Ton, L.-S. Lê, and H.-A. Pham, "Towards a domain specific framework for wearable applications in Internet of Things," in *Proc. 4th Int. Conf. Future Data Secur. Eng.* Ho Chi Minh City, Vietnam: Springer, Nov. 2017, pp. 309–324.

[47] L.-P. Ton and L.-S. Lê, "Enacting a rule-based alert business process in smart healthcare using IoT wearables," in *Proc. IEEE 23rd Int. Enterprise Distrib. Object Comput. Workshop (EDOCW)*. Paris, France: IEEE Computer Society, Oct. 2019, pp. 104–107.

[48] R. Sinha, A. Narula, and J. Grundy, "Parametric statecharts: Designing flexible IoT apps: Deploying Android m-health apps in dynamic smart-homes," in *Proc. Australas. Comput. Sci. Week Multiconference*. Geelong, VIC, Australia: ACM, Jan. 2017, pp. 1–8.

[49] J. Beard, "State machines as a service: An SCXML microservices platform for the Internet of Things," in *Proc. 2nd Workshop Eng. Interact. Syst. SCXML, Conjunct. (EICS)*. Duisburg, Germany: ACM, Jun. 2015, pp. 17–21.

[50] P. Garamvolgyi, I. Kocsis, B. Gehl, and A. Klenik, "Towards model-driven engineering of smart contracts for cyber-physical systems," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*. Luxembourg, Europe: IEEE Computer Society, Jun. 2018, pp. 134–139.

[51] C. Guo, Z. Fu, Z. Zhang, S. Ren, and L. Sha, "Design verifiably correct model patterns to facilitate modeling medical best practice guidelines with statecharts," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 6276–6284, Aug. 2019.

[52] E. Domínguez, B. Pérez, Á. L. Rubio, and M. A. Zapata, "A systematic review of code generation proposals from state machine specifications," *Inf. Softw. Technol.*, vol. 54, no. 10, pp. 1045–1066, Oct. 2012.

[53] S. Bonfanti, A. Gargantini, and A. Mashkoor, "Design and validation of a C++ code generator from abstract state machines specifications," *J. Softw., Evol. Process*, vol. 32, no. 2, pp. 2205–2232, Feb. 2020.

[54] A.-J. Salman, M. Al-Jawad, and W. A. Tameemi, "Domain-specific languages for IoT: Challenges and opportunities," in *Proc. 4th Int. Conf. Eng. Sci.* Kerbala, Iraq: IOP, Dec. 2020, pp. 121–133.

[55] C. G. García, D. Meana-Llorián, B. C. P. G-Bustelo, J. M. C. Lovelle, and N. Garcia-Fernandez, "Midgar: Detection of people through computer vision in the Internet of Things scenarios to improve the security in smart cities, smart towns, and smart homes," *Future Gener. Comput. Syst.*, vol. 76, pp. 301–313, Nov. 2017.

[56] P. Arcaini, S. Bonfanti, A. Gargantini, E. Riccobene, and P. Scandurra, "Modelling an automotive software-intensive system with adaptive features using ASMETA," in *Proc. Int. Conf. Rigorous State-Based Methods*. Ulm, Germany: Springer, 2020, pp. 302–317.

[57] S. Bonfanti, A. Gargantini, and A. Mashkoor, "Validation of transformation from abstract state machine models to C++ code," in *Proc. IFIP Int. Conf. Test. Softw. Syst.* Paris, France: Springer, 2018, pp. 17–32.

[58] Y. Chen and G. De Luca, "VIPLE: Visual IoT/robotics programming language environment for computer science education," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*. Chicago, IL, USA: IEEE, May 2016, pp. 963–971.

[59] M. Balsiger, "A quick-start tutorial to eclipse plug-in development," Inst. Comput. Sci. Appl. Math., Univ. Bern, Bern, Switzerland, Tech. Rep., Dec. 2010.

[60] A. Wegmann, L.-S. Lê, L. Hussami, and D. Beyer, "A tool for verified design using alloy for specification and CrocoPat for verification," in *Proc. 1st ACM Alloy Workshop*. Portland, OR, USA: ACM, Nov. 2006, p. 58.

LONG-PHUOC TÔN received the B.Sc. degree from the University of Science, VNU-HCM, Vietnam, in 2001, and the M.Sc. degree from the University of Information Technology, VNU-HCM, in 2006. He is currently pursuing the Ph.D. degree in computer science with the Ho Chi Minh City University of Technology, VNU-HCM. He is currently a Lecturer with the Industrial University of Ho Chi Minh City (IUH). His Ph.D. work is theoretically focused on the engineering of a domain-specific language and a toolkit for the IoWT programming.

LAM-SON LÊ (Member, IEEE) received the Ph.D. degree from the EPFL, Switzerland, in 2008. He held a postdoctoral position with the University of Wollongong, Australia, for four years. He currently holds an academic position with the Faculty of Computer Science and Engineering, HCMC University of Technology, VNU-HCM. He leads a research group on data-driven enterprise software and processes. He has authored and coauthored approximately 80 scientific articles published in the proceedings of highly ranked conferences and in reputable journals in the fields of enterprise computing, business processes management, service-oriented computing, conceptual modeling, and information systems engineering. He served as the conference chair and a member for the technical program committee of multiple international conferences on enterprise computing and advanced computing for digital transformation.

MINH-SON NGUYEN received the B.Engr. and M.Engr. degrees in computer engineering from the HCMC University of Technology, Vietnam, in 2001 and 2005, respectively, and the Ph.D. degree (Hons.) in electrical engineering from the University of Ulsan, South Korea, in 2010. He currently serves as the Dean of the Faculty of Computer Engineering, University of Information Technology–Vietnam National University, Ho Chi Minh, where he is also the Director of the Automotive Research and Development Laboratory. He is a member of the Committee of Science and Technology of Saigon High-Tech Park. His research interests include wireless embedded internet, AI for the IoT, smart systems, and system-on-chip design.