Interoperable Real-Time Symbolic Programming for Smart Environments

Article

Accepted version

S. Gaglio, G. Lo Re, L. Giuliana, G. Martorella, D. Peri, A. Montalto

In Proceedings of IEEE International Conference on Smart Computing (SMARTCOMP), Washington, DC, USA, 2019, pp. 309-316

Interoperable Real-Time Symbolic Programming for Smart Environments

Salvatore Gaglio*, Giuseppe Lo Re†, Leonardo Giuliana‡, Gloria Martorella§,
Antonio Montalto¶, Daniele Peri‖
University of Palermo, Department of Engineering, Viale delle Scienze, Ed. 6, Palermo, Italy
Email: *salvatore.gaglio@unipa.it, †leonardo.giuliana@community.unipa.it, ‡giuseppe.lore@unipa.it,
§gloria.martorella@unipa.it, ¶antonio.montalto@community.unipa.it, ‖daniele.peri@unipa.it

***Abstract*—Smart environments demand novel paradigms offering easy configuration, programming and deployment of pervasive applications. To this purpose, different solutions have been proposed ranging from visual paradigms based on mashups to formal languages. However, most of the paradigms proposed in the literature require further external tools to turn application description code into an executable program before the deployment on target devices. Source code generation, runtime upgrades and recovery, and online debugging and inspection are often cumbersome in these programming environments.**

In this work we describe a methodology for real-time and on-line programming in smart environments that is compact and efficient enough to run on resource-constrained devices. The pillar of the proposed approach is real-time exchange of executable symbolic code in heterogeneous networks. The methodology is supported by an inference engine that is able to generate symbolic code starting from knowledge about hardware devices and their placement in the environment, and about the application domain.

Interoperability with existing smart applications and Internet of Things (IoT) deployments is reached through a symbolic Transmission Control Protocol (TCP) client, and Message Queue Telemetry Transport (MQTT) client.

***Index Terms*—Symbolic processing, Knowledge base, Executable code exchange, Forth, MQTT, Resource-constrained devices.**

I. Introduction

Smart environments rely on many pervasive devices supporting users at home [1], office [2], and in urban [3] contexts by offering a wide range of services and possibilities that include automation and reasoning [4].

But as provided services and smartness increase, so does the complexity of the applications and the factors at stake during all the development phases. For instance, in smart cities the huge amount of real-time updates makes (re-)configuration of a large number of devices at execution time particularly laborious [5]. Significant efforts are demanded to modify application logic running on several smart devices in real-time–e.g. for upgrades or debugging–or to change the usage scenario. Online reprogramming must be carried out without physical wired access to smart devices. Indeed, Cloud-based solutions have been adopted with the entire application running outside the network on powerful servers [3]. Further issues emerge whether resource-constrained devices must interoperate with resource-rich elements that support a wide variety of protocols [6].

In this context, several computing paradigms and frameworks have been proposed to reduce the effort required for configuration, implementation and reprogramming of smart environments. Visual approaches offer fast configuration and programming through basic composition of blocks that represent predefined functions [7], [8]. Effective and complete customization is not possible at all as blocks are predefined, difficult to extend, and possibly not particularly suitable for novice users [9]. Web-based paradigms explore standard technologies and protocols in semi-automatic and automatic service mashups [10]. However, static mashups are not suitable for highly dynamic scenarios such as smart environments [4]. On the contrary, the presence of inference engines for dynamic mashup generation often involves different levels of translation to obtain the code that actually runs on the devices [11]. Although trigger-action schemes offer intuitive design approaches through condition/action specifications [12], some limitations occur for the implementation of more advanced applications, e.g. those involving on-board reasoning. Formal paradigms allow to configure and program smart environments correctly by providing methods for tracking errors and correcting them in the early stages of development [13]. Nevertheless, programming expertise is needed to express high-level applications through formal languages. Metamodels within the agent-oriented programming paradigm have also been adopted [14] in all the application development stages [15]. Abstractions provided by high-level metamodels are progressively lost during their refinements as well as the binding between descriptions and implementation code. Most of the proposed programming paradigms target specific application domains and interoperability is hindered by easily arising incompatibility issues [5].

In this paper we address easy configuration, programming, and updates of smart environments by exploiting a symbolic programming paradigm. The main feature of the proposed methodology lies on real-time exchange of symbolic code among nodes. We present a framework which integrates an inference engine that automatically generates symbolic code to configure, query and program smart nodes in real-time. We also propose a symbolic platform running on resource-constrained smart devices that supports distributed applications through executable code exchange. The implementation of symbolic Transmission Control Protocol (TCP) and Message

Queue Telemetry Transport (MQTT) clients represents a further step towards interoperability.

The paper is organized as follows. Most important paradigms targeting smart environments are presented in Section II, while the methodology based on the symbolic programming model is outlined in Section III. Section IV details the framework based on executable symbolic code exchange among smart devices. The rule-based system and the symbolic platform are presented in Section V and in Section VI, respectively. The experimental evaluation of the symbolic platform running on nodes is described in Section VII. Finally, conclusions and future work are reported in Section VIII.

II. Related Work

A large effort to make configuration, programming and deployment of smart applications more accessible and interoperable can be found in the literature. Indeed, several frameworks and many computing models have been proposed.

Visual paradigms have been adopted to enable novice users designing their own smart application without requiring any expertise. For instance, a graphical programming paradigms that consists of wiring a set of predefined blocks, each of which represents a function was proposed [9]. An inference engine is responsible of translating user-defined applications to Python scripts. The code is then deployed to nodes at runtime. A set of steps for code translations are not confined to source code but also to communication protocols. In fact, a more powerful device, i.e. a proxy server, converts HTTP requests to CoAP, a Web protocol for resource constrained devices. Visual interface tools present some limitations in terms of both application and network customization. In fact, blocks are defined in advance and their extension quite impracticable. A process-oriented paradigm for smart buildings was also proposed [16]. It is supported by a process-oriented Domain Specific Language (DSL) that provides high-level abstractions to orchestrate smart objects. A semi-automatic approach is adopted as the programmer defines the control workflow, while devices are discovered automatically through a central unit. High-level specifications are defined through the use of ontologies, whose porting and updating on board resource-constrained devices appears a rather impracticable solution. A Web-based framework embodying dynamic mashup generation was presented for high dynamic scenarios, such as smart environments [11]. Provided with a high-level user goal and a set of available services, the framework infers user actions in the form of API requests to be executed. A trigger-action programming model targeting smart application was also adopted [12]. Application development is done through a condition/actions approach which is often supported by visual editors.

Agent-oriented computing was exploited to model distributed software systems in terms of multiagent systems (MASs) [15] . The methodology consists in the production of different metamodels, one for each of the development phases (analysis, design and implementation). Each metamodel is refined in the successive phase. Finally, the implementation metamodel is used to generate the implementation code targeting the Jade platform. However, transitions among metamodels are not automatic.

A formal approach using π-calculus was proposed to program complex smart systems that prove correct [13]. Smart environments are represented as π-calculus statements and then this representation traverses a set of transformation phases to produce Java source code. Although verification is taken into account, the support to configuration, reprogramming and code deployment is still lacking.

III. Symbolic Programming Paradigm

The symbolic programming paradigm we adopted is rooted into the concept of *word*, a symbol that intrinsically represents a computation and which is defined as a chain of other, previously defined, words. This software abstraction exhibits a tree structure as the execution of a word involves executing all the words composing its definition, that in turn are defined in terms of other words, and so forth, until a leaf symbol is reached. As words are user-defined, no limits are imposed and word names can be taken directly from the natural language vocabulary so that a strict association with the domain under consideration could emerge clearly. Indeed, this paradigm concretely supports high-level concepts and abstractions, physical world semantics, easy design and development of applications.

For instance the following phrase:

```
LAMP ON
```

is an application that actuates on the environment turning the lamp on, while:

```
LAMP BLINK 3 TIMES
```

is an application that blinks a lamp three times.

Extending this concept, smart environments can be thus supported by defining words for sensing, actuating and reasoning.

Besides running on resource-rich devices that can support thick software architectures, including a symbolic interpreter, this paradigm runs also efficiently on the bare hardware, with compact memory footprint [17]. This makes it suitable for enabling symbolic processing on resource-constrained devices and paves the way to make them interoperable with other heterogeneous and more powerful devices. The symbolic model is concretely supported by Forth, a stack-based programming language which provides both interactive and compiled execution modes.

IV. System Overview

In the following, we describe the framework we implemented that exploits the symbolic methodology introduced in the previous section. The proposed system is composed of:

- A rule-based system that holds specifications about the physical environment, hardware platforms and about the high-level smart application. A backward chaining inference engine automatically generates symbolic code for runtime configuration, programming or updating already deployed devices.

- A software platform running on smart devices that makes them able to process and exchange symbolic code. The platform also includes (i) a symbolic TCP client for the integration with existing Internet-based infrastructures and (ii) an MQTT-based client enabling interoperability with IoT-based applications.
  The rule-based system is implemented in Prolog and runs on a host while the smart devices are based on the widespread low-cost Wi-Fi enabled ESP8266 system on a chip.

V. Automatic Configuration and Programming Tool

The rule-based system is composed of two main blocks: (i) a knowledge base provides a formal representation of all of the aspects of a smart environment scenario, as a whole (ii) an inference engine that automatically (re-)configures and (re-)programs already deployed smart devices by sending them symbolic code. Functionally, the knowledge base is organized in four rule sets. The *Physical World* rule set models facts and rules that are related to the physical environment, which is formally specified through physical quantities and states. This layer is independent of a specific use context and can be exploited by multiple domains. The *Application* rule set holds the environmental description that is strictly related to high-level applications. Specifications include a set of objects, their locations and states. Locations refer to generic or specific object placements, while objects that are deployed in the environment, and that can be managed by the system, are also described. Each location can contain sub-locations, proceeding from general to specific ones. A location thus exhibits a tree structure, from a root node to leaf nodes. For instance, the root location "Home" is composed of other locations, e.g. "Room" and "Kitchen". Finally, leaf nodes identify real-world object positions, as shown in Figure 1.

Hardware platforms are modeled in the *Hardware* rule set which provides descriptions from a low level point of view. This component includes MCUs, their pins, peripherals, sensors and actuators, as specified by the technical documentation or data-sheets. Finally, the *Network* rule set models the channel to manage the remote connection between each device and the rule-based system for configuration and code transmission. Concepts as host name, IP address, server port number are also included. Provided with this body of facts and rules and a high-level task given by the user, the rule-based system acts as a *Configurator* inferring the symbolic code for device configuration and for the actions required to match the user-defined goal. The symbolic code is sent to smart devices as a sequence of textual strings. In this work, configuration refers both to the connection of devices with the system, i.e. the device installation at bootstrap, and the interconnection of sensors and actuators to a specific MCU, i.e. the best way of connecting sensors and actuators considering the available I/O pins on the devices.

To send the code, the configurator opens a TCP connection to the TCP-REPL listen port of the receiver device. After the connection has been established, the code is sent via TCP to the remote node.

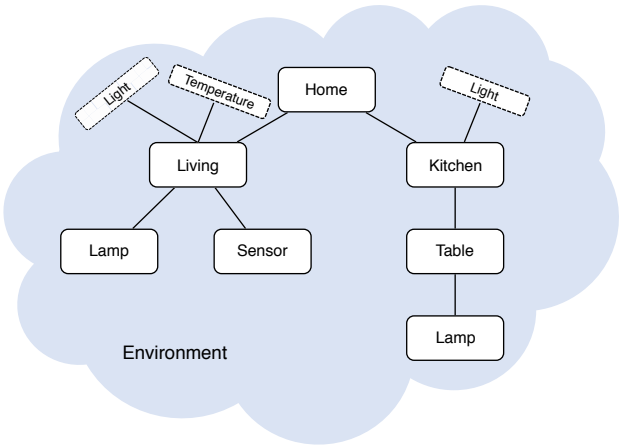

Fig. 1: Tree structure of the physical environment as a set of locations and objects.

As an example, in the knowledge base an instance of the ESP8266_12E board is defined as such:

```
mcu_name(myesp, esp8266_12e)
```

that associates the unique label myesp to the hardware platform ESP8266_12E. Its IP address and TCP-REPL port are specified as such:

```
mcu_net_address(myesp,
'myesp.local', 1983).
```

Through the mcu_send_message procedure, the configurator connects to the remote TCP-REPL server of myesp and sends it the symbolic code to make the RLED LED be turned on and then off with a 500 ms time interval:

```
mcu_send_message(myesp,
['RLED ON 500 ms RLED OFF']).
```

Considering realtime code generation, the execution of:

```
exec(home, lamp, on).
```

generates the code to turn all of the home lamps on, while to read the unique luminosity sensor at home:

```
execr(home, light, check, A).
```

In this case the variable `A` stands for 'answer' and holds the luminosity sensor value. The generated symbolic code is:

```
LUMSENS READ .
```

The word `.` displays the answer. The organization of the rule-based system and the code generation process is outlined in Figure 2.

VI. Symbolic Programming on Resource-Constrained Devices

The interpreter for symbolic code is the key component of the operating environment we propose for smart resource-

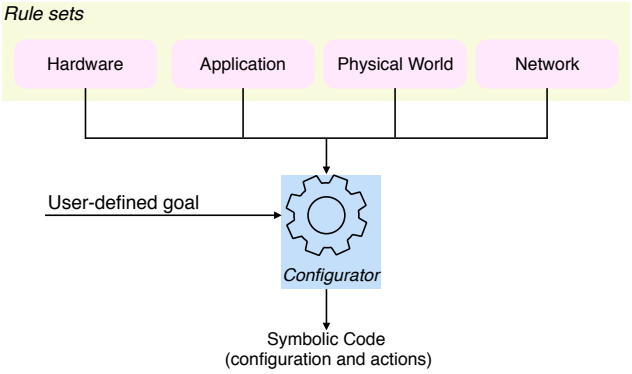

Fig. 2: Structure of the rule-based system and code generation process.

constrained devices. The interpreter executes words in real-time on the target hardware so that a cross-compilation toolchain is not needed. To enable the execution of symbolic code by a remote device, the platform includes a TCP-REPL server which waits for incoming symbolic code. The symbolic platform we implemented is built atop PunyForth [18], a simple Forth-based programming environment that implements the computing paradigm described in Section III. The symbolic interpreter is an application running above the node operating system. A wrapper exposes network and hardware access functionalities, which are implemented in the operating system SDK, through a high-level symbolic language. Further functionalities are available in the form of modules that can be loaded as needed.

A *dictionary*, which is extensible interactively, stores all the executable symbols, i.e. words. This reflects on the fact that device skills or their tasks, can be extended at runtime without great efforts. For instance, to equip an already operating device with a red LED, the generated symbolic code for configuration looks like this:

```
: HIGH GPIO_HIGH gpio-write ;
: LOW GPIO_LOW gpio-write ;
: SET_MODE gpio-mode ;
: ON HIGH ;
: OFF LOW ;

12 CONSTANT RLED
RLED GPIO_OUT SET_MODE

store
```

The execution of the word `:` begins a word definition. It is followed by the name of the new word and the chain of words composing it. The word `;` ends the definition. The configurator generates all the required definitions as well as the suitable hardware configuration, i.e. the red LED has to be connected to the pin 12. Indeed, the symbol RLED is defined as a constant whose value is the number 12. The word `ON` implies executing the word `HIGH`, which is in turn defined as the set of words to put the GPIO high. Finally, the word `store` writes the configuration in the Flash memory of the node. Once the node has been configured, the following code is generated and sent for the runtime execution of the desired reactive behavior, i.e. blinking the LED:

```
RLED ON 500 ms RLED OFF
```

On remote devices, the TCP-REPL server listens for client connections. The server operation is split into two phases: storage and interpretation. First, it reads the received characters from the listening socket and stores them in a buffer. Once the sequence `CR LF` is encountered, the word `eval` starts the interpretation phase. The buffer is then released. Once the word `quit` is received and interpreted, the connection between sender and receiver is closed. Finally, the interpretation of the word `undo` causes flashing the buffer even if the terminator sequence has not been received yet. The code exchange between the configurator and the device is depicted in Figure 3.

Listing 1: Symbolic code for enabling a TCP client on smart devices.

```
0 init-variable: socket
0 task: netfetch-task
128 buffer: line

: tcp-receive
  activate
    println: "Start"
    begin
      socket @ 128 line netcon-readln
          -1 <>
      line "quit" =str invert and
    while
      line strlen if line eval then
    repeat
    drop socket @ netcon-dispose
    0 socket !  println: "End"
        deactivate ;

: tcp-send
  socket @ if
    socket @ swap netcon-writeln
  then ;

: tcp-disconnect
  "quit" tcp-send ;

: tcp-connect ( port ip -- )
  multi
  0 socket !
  TCP netcon-connect socket !
  netfetch-task tcp-receive ;

: tcp
  ['] tcp-connect
  ['] tcp-disconnect
  ['] tcp-send ;
end
```

Our platform includes: (i) a symbolic TCP client that enable executable code exchange among peer nodes to support distributed smart applications (ii) a lightweight MQTT [19] client for easy integration and interoperability with existing IoT deployments.

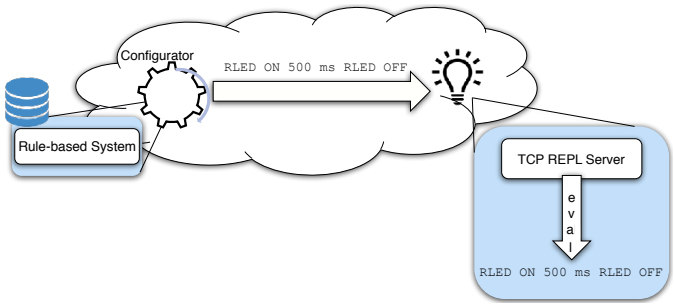

Fig. 3: Interaction between the configurator and a deployed node. The symbolic code sent by the configurator to the TCP REPL server to make the LED blink is reported above the horizontal arrow.

TABLE I: Summary table of low-level words used to implement the TCP client

Word	Description
`tcp-connect`	Open a TCP connection to the specified client and the respective socket. The task responsible of listening for incoming characters is started.
`tcp-send`	Send the code via TCP
`tcp-receive`	Read incoming chars from the listen socket and execute the word `eval` to start the interpretation.
`tcp-disconnect`	Close the TCP connection between two nodes.
`tcp`	Puts on the stack the addresses of all the words described above, except for `tcp-receive`.

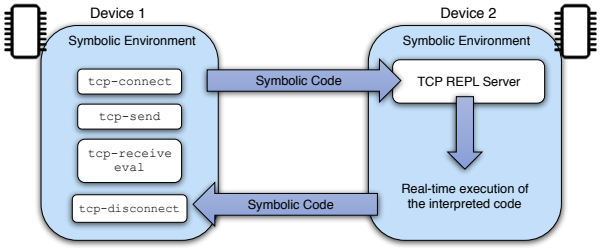

Fig. 4: Interaction scheme between two smart devices. Device 1 sends symbolic code via TCP to Device 2 that listens on the TCP-REPL port, then the Device 2 can close the TCP connection.

Multitasking is also supported. A client MQTT and a server REPL are, in fact, simultaneously active.

The overall architecture of the proposed platform is depicted in Figure 5.

A. Executable Code Exchange among Smart Devices

Unidirectional symbolic code exchange, from the configurator to target devices, is not the unique operating mode supported by our platform. Indeed, we implemented a symbolic TCP client on target devices in a few code lines. This way, each node is able to receive and execute symbolic code through the TCP-REPL server and to send it to other devices through the TCP client. Such an interaction scheme among smart devices is illustrated in Figure 4. In our environment, a complete TCP client is quite compact as the implementation requires the definition of just four words, as reported in Listing 1. A summary of defined words along with their descriptions are provided in Table I.

For instance, to make a device with address 192.169.4.4 connect to the device with address 192.169.4.2 , the symbolic code to start a TCP connection on port 1983, which is the TCP-REPL server listen port, is:

```
1983 "192.168.4.2" tcp connect
"CONF load\r\n" tcp send
"RLED ON\r\n" tcp send
tcp disconnect
```

The word `tcp` acts as a protocol specifier. Therefore, more generic high-level words have been also included in the dictionary to operate with different protocols. For instance, the word `connect`, discards the addresses of `tcp-send`, `tcp-disconnect` left by the word `tcp` and executes the word `tcp-connect`. A similar behavior is performed by the word `send` that sends the opportune configuration and the symbolic code to make the RLED on via TCP. Finally, the word `disconnect` sends the string "quit" that closes the TCP connection between the nodes.

Enabling executable code exchange of symbolic code among smart devices really makes smart devices independent from any central entity, including the configurator and opens up news possibilities. In fact, a node can extend the dictionary of a remote node, query another device placed in the environment, send the code to start a collaborative behavior or to reprogram it and so on. The proposed mechanism thus provides concrete support for the design, implementation and deployment of distributed smart applications.

B. Support to Interoperability in Smart Environments

Smart environments are integrated within IoT scenarios using high-level application protocols that enable cooperation of devices and technologies. Among these, MQTT [19] is a *de facto* standard for interoperability among heterogeneous devices through a publish/subscribe interaction scheme. Certain devices act as topic publishers while others as subscribers to a certain topic. The entire operation is managed by the broker, a central entity that dispatches messages between publishers and subscribers. To make our platform interoperable with existing IoT applications, the platform supports symbolic code exchange over MQTT. Different MQTT implementations are available as C libraries (ESP-RTOS-MQTT) that result

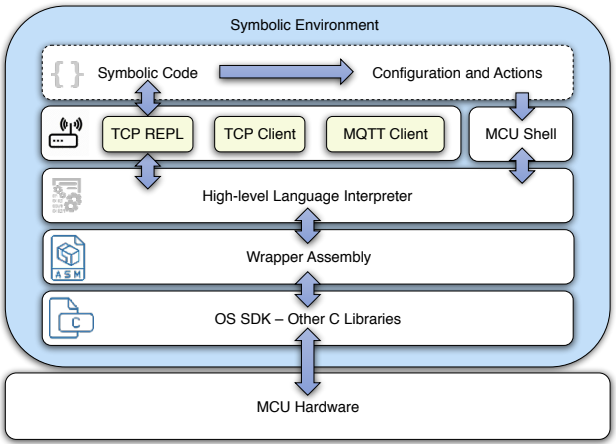

Fig. 5: Architecture of the proposed platform.

particularly structured and their compilation quite involved for the target hardware platform we adopted.

Instead, we developed a symbolic MQTT client in a few lines of code that does not require compilation and reflashing, as the code is interpreted interactively. Although, the implementation is not complete, it works as expected. According to the protocol specification, the environmental description and device placement follow a hierarchical structure, as that adopted by the configurator and described in Section V. For instance, a possible topic is:

```
home/kitchen/table/lamp/1
```

The main advantage of a symbolic implementation is primarily code compactness and incremental development. Indeed, compilation and reflashing of the whole binary node image are completely avoided as the code is executed interactively. The list of words and their descriptions are provided in Table II

A possible scenario consists in two ESP8266 devices, named `esp1` and `esp2`, provided with a red LED and a green LED respectively. An Intel x86-64 server running Fedora Linux OS named vls1 with IP address 192.168.0.19 runs the open-source MQTT broker mosquitto. The server also runs the mosquitto_pub MQTT publisher application. The user can interact with deployed devices through a simple telnet terminal. For instance, the user can connect to the `esp1` TCP-REPL server and send it the symbolic code to make it register to a specified topic on the broker in realtime:

```
1883 "192.168.0.19" mqtt connect
"home/kitchen/table/led/red" mqtt send
```

A similar code can be sent to `esp2`, as follows:

```
1883 "192.168.0.19" mqtt connect
"home/kitchen/table/led/green" mqtt send
```

To switch on all the LEDs that are placed on the table, the publisher sends the code `LED ON` over MQTT to all subscribers as follows:

TABLE II: Summary table of low-level words used to implement the TCP client

Word	Description
`mqtt-connect`	Connect to a broker using IP e PORT already on the stack by sending a CONNECT message.
`mqtt-publish`	Given a certain message and a topic, build a PUBLISH message and send it to the broker.
`mqtt-subscribe`	Read incoming chars from the listen socket and execute the word `eval` to start the interpretation.
`mqtt-disconnect`	Send a DISCONNECT message to the broker and close the socket.
`mqtt-ping`	Build and send a PINGREQ message to the broker.
textttmqtt-ping-worker	Execute the word `mqtt-ping` every 30 seconds.
`mqttread`	Read message sent by the broker and store it in a buffer.
`mqttread-worker`	Read message sent by the broker and store it in a buffer and perform the display on the output stream.
`mqtt`	Puts on the stack the addresses of all the words described above, except for `mqttread`.

```
mosquitto_pub -h 192.168.0.19 -p 1883 -t
"home/kitchen/table/led/#" -m "LED ON"
```

This interaction scheme described above is depicted in Figure 6.

VII. Experimental Evaluation

In the following, we describe the experimental tests we carried on to evaluate our framework in terms of memory footprint and code compactness. The target platform used for the experimental evaluation is the ESP8266-12E, which includes a 32-bit RISC processor, a 4 MB external Flash, 50 KB of available RAM, and integrates a WI-Fi interface and a complete TCP-IP stack. The ROM is not programmable while the user program is stored in the external Flash. The board has 22 pin that expose several interfaces, such as I2C, SPI, UART and GPIO. Considering the complete platform implementation, which include both the TCP and MQTT clients, binary files and symbolic code files were loaded into the Flash memory. Their respective occupation is reported in Table III. Finally, the Flash footprint amounts to 388096 B.

The RAM occupation has been measured through the words `osfreemem`, `usedmem` e `freemem` which were already available in Punyforth. These words display the available memory for the operating system, the memory occupied in the heap and the available memory in the heap, which is reserved to Punyforth. A RAM description from a quantitative point of view if found in Table IV. As shown, the available RAM for new definitions amounts to 6204 B.

We also evaluated the footprint of the binary image as well as occupation of source files (Figure 7).

As a first step, the binary image of the operating system and SDK has been measured. To the purpose, an application consisting in a void loop has been compiled and the binary

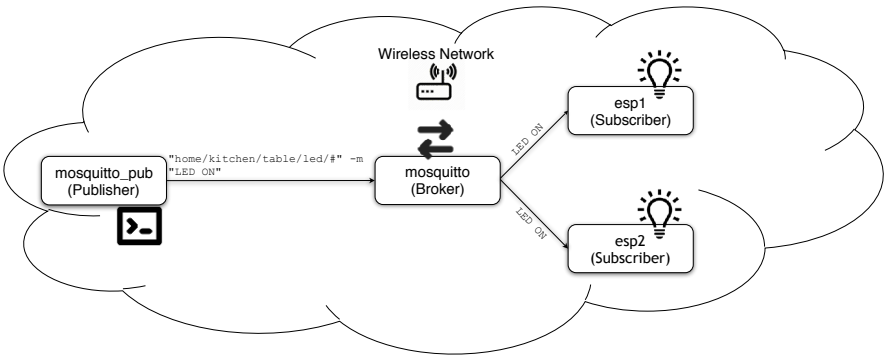

Fig. 6: Symbolic code exchange among smart devices over MQTT

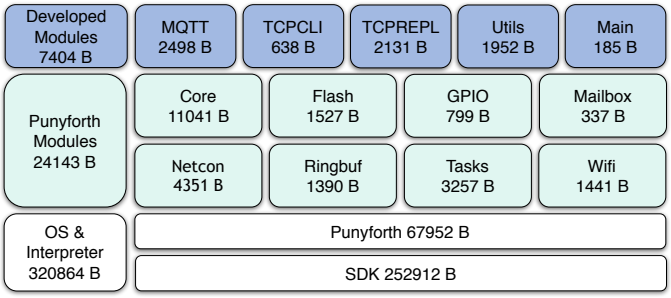

Fig. 7: Architecture of the proposed platform and memory occupation of each component.

TABLE III: Flash occupation of binary and symbolic code files loaded into the Flash memory.

Source	Flash Address	Footprint [B]
`rboot.bin`	0x00	4096
`blank_config.bin`	0x1000	2048
`punyforth.bin`	0x2000	321536
`main.forth`	0x52000	1024
`modules.foth`	0x53000	59392

TABLE IV: Quantitative RAM description

RAM area	Footprint [B]
Available RAM for OS	14204
Occupied RAM within the heap	28980
Available RAM within the heap	6204

image size has been measured. The difference between the obtained result and the image of the SDK, previously computed, provides the actual image size of PunyForth.

Considering the symbolic MQTT client, we compared it to the a C-based MQTT implementation (https://github.com/vaibhav93/ESP-RTOS-MQTT) assessing code compactness in terms of lines of code and number of chars. Such a metric provides an estimate about the complexity of source code development, as well as about debug effort since the number of possible errors and bugs rises as the lines of code increase. Results reported in Table V show that the symbolic version is more compact than the correspondent C implementation.

We also assessed the turnaround time required for symbolic

TABLE V: Comparison between C-based MQTT and symbolic MQTT

MQTT Implementation	LOC	Number of chars
`C-based`	3220	97006
`Symbolic`	132	2420

TABLE VI: Turnaround time for symbolic code transfer over TCP

Transferred byte	Minimum Time [s]	Maximum Time [s]	Average Time [s]
352	0.0057	0.1938	0.5226
697	0.0775	0.2229	0,5325
1387	0.1043	0.1982	0.5557
2767	0.1503	0.1903	0.5862

code transfer from a TCP client to a TCP-REPL server. Both the TCP client and the receiver has been connected to the same Access Point via Wi-Fi. The test has been carried out by sending the same symbolic code 100 times. The number of bytes sent has been progressively doubled so that 4 test sessions were completed for a total of 400 code transfers. Finally, minimum, maximum and average time are reported as the number of bytes increases in Table VI.

VIII. Conclusions

This paper presented symbolic programming as an effective paradigm for easy (re-)configuration and (re-)programming of smart applications. The methodology is exploited by a framework that is composed by: (i) an inference engine for runtime automatic symbolic code generation and sending and (ii) a symbolic platform that allows node to execute incoming code at runtime. Symbolic code exchange is also enabled between deployed smart devices, no matter the presence of the inference engine. To support interoperable code exchange, symbolic TCP and MQTT clients were developed. Three main advantages arise from adopting symbolic programming and its exchange: (i) high-level processing is enabled even on resource constrained- devices. This reveals an interoperable solution allowing the application logic be located in the network as resource-constrained devices and more powerful nodes can process and exchange symbolic code; (ii) high-level code can be directly executed on target hardware without any intermediate translation phase. The high-level application

code is the same on different devices, while only the low-level coding of each symbol is specific to the underlying hardware; (iii) symbolic code exchange can include code involving partial changes, hardware inspection, extension of node skills, configuration, fuzzy rules [20] and so on. Experimental results confirm the feasibility of the approach. Future work will include the implementation of a symbolic broker, the possibility of storing new word definition on the Flash, and a garbage collector to free up and use RAM areas.

REFERENCES

[1] M. Alaa, A. Zaidan, B. Zaidan, M. Talal, and M. Kiah, "A review of smart home applications based on internet of things," *Journal of Network and Computer Applications*, vol. 97, pp. 48 – 65, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804517302801

[2] A. Fernandez-Montes, J. Ortega, J. Sanchez-Venzala!', and L. Gonzalez-Abril, "Software reference architecture for smart environments: Perception," *Computer Standards & Interfaces*, vol. 36, no. 6, pp. 928 – 940, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0920548914000300

[3] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "Fogflow: Easy programming of iot services over cloud and edges for smart cities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 696–707, April 2018.

[4] S. Mayer, N. Inhelder, R. Verborgh, R. V. de Walle, and F. Mattern, "Configuration of smart environments made simple: Combining visual modeling with semantic metadata and reasoning," in *2014 International Conference on the Internet of Things (IOT)*, Oct 2014, pp. 61–66.

[5] E. F. Z. Santana, A. P. Chaves, M. A. Gerosa, F. Kon, and D. S. Milojicic, "Software Platforms for Smart Cities: Concepts, Requirements, Challenges, and a Unified Reference Architecture," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 78:1–78:37, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3124391

[6] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, Fourthquarter 2015.

[7] M. Seraj, S. Autexier, and J. Janssen, "Beesm, a block-based educational programming tool for end users," in *Proceedings of the 10th Nordic Conference on Human-Computer Interaction*, ser. NordiCHI '18. New York, NY, USA: ACM, 2018, pp. 886–891. [Online]. Available: http://doi.acm.org/10.1145/3240167.3240239

[8] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, *From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 97–129. [Online]. Available: https://doi.org/10.1007/978-3-642-19157-2_5

[9] M. A. Serna, C. J. Sreenan, and S. Fedor, "A Visual Programming Framework for Wireless Sensor Networks in Smart Home Applications," in *2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, April 2015, pp. 1–6.

[10] R. Kleinfeld, S. Steglich, L. Radziwonowicz, and C. Doukas, "Glue.things: A mashup platform for wiring the internet of things with the internet of services," in *Proceedings of the 5th International Workshop on Web of Things*, ser. WoT '14. New York, NY, USA: ACM, 2014, pp. 16–21. [Online]. Available: http://doi.acm.org/10.1145/2684432.2684436

[11] S. Mayer, R. Verborgh, M. Kovatsch, and F. Mattern, "Smart Configuration of Smart Environments," *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 3, pp. 1247–1255, July 2016.

[12] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, "Practical Trigger-action Programming in the Smart Home," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 803–812. [Online]. Available: http://doi.acm.org/10.1145/2556288.2557420

[13] V. G. Lekshmy and J. Bhaskar, "Programming smart environments using p-calculus," *Procedia Computer Science*, vol. 46, pp. 884 – 891, 2015, proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace and Island Resort, Kochi, India. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915002227

[14] F. Cicirelli, G. Fortino, A. Guerrieri, G. Spezzano, and A. Vinci, "Metamodeling of smart environments: from design to implementation," *Advanced Engineering Informatics*, vol. 33, pp. 274 – 284, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474034616302063

[15] G. Fortino, W. Russo, C. Savaglio, W. Shen, and M. Zhou, "Agent-Oriented Cooperative Smart Objects: From IoT System Design to Implementation," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 48, no. 11, pp. 1939–1956, Nov 2018.

[16] A. Albreshne and J. Pasquier, "A domain specific language for high-level process control programming in smart buildings," *Procedia Computer Science*, vol. 63, pp. 65 – 73, 2015, the 6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2015)/ The 5th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2015)/ Affiliated Workshops. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915024412

[17] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "DC4CD: A Platform for Distributed Computing on Constrained Devices," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 1, pp. 27:1–27:25, 2018. [Online]. Available: http://doi.acm.org/10.1145/3105923

[18] "PunyForth: Forth inspired Programming Language for the ESP8266," https://github.com/zeroflag/punyforth, online; accessed: 2019-01-25.

[19] O. Standard, "Message Queue Telemetry Transport Documentation," http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf, 2014, online; accessed 06 June 2018.

[20] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices," *EAI Endorsed Transactions on Cognitive Communications*, vol. 15, no. 2, 5 2015.