Stefano Scanzio[1], Gianluca Cena[1], and Adriano Valenzano[1]

[1]National Research Council of Italy (CNR-IEIIT)

April 09, 2024

This is the author's version of an article that has been published in this journal.
Changes were made to this version by the publisher prior to publication.
The final version of record is available at https://doi.org/10.1109/ETFA52439.2022.9921530

# QRscript: Embedding a Programming Language in QR codes to support Decision and Management

Stefano Scanzio, Gianluca Cena, and Adriano Valenzano
National Research Council of Italy (CNR–IEIIT), Corso Duca degli Abruzzi 24, I-10129 Torino, Italy
Email: {stefano.scanzio, gianluca.cena, adriano.valenzano}@ieiit.cnr.it

***Abstract*—Embedding a programming language in a QR code is a new and extremely promising opportunity, as it makes devices and objects smarter without necessarily requiring an Internet connection. In this paper, all the steps needed to translate a program written in a high-level programming language to its binary representation encoded in a QR code, and the opposite process that, starting from the QR code, executes it by means of a virtual machine, have been carefully detailed. The proposed programming language was named QRscript, and can be easily extended so as to integrate new features.**

**One of the main design goals was to produce a very compact target binary code. In particular, in this work we propose a specific sub-language (a dialect) that is aimed at encoding decision trees. Besides industrial scenarios, this is useful in many other application fields. The reported example, related to the configuration of an industrial networked device, highlights the potential of the proposed technology, and permits to better understand all the translation steps.**

***Index Terms*—QR code, decision trees, compilers, management, maintenance, QRscript.**

## I. Introduction

World is changing, digital networks and connectivity are more and more pervasive, and communication between intelligent objects is nowadays extremely common, to the point that this trend was explicitly given a specific name, that is, the Internet of Things (IoT) [1]. Industry is changing too, and paradigms like Industry 4.0 [2] and the Industrial IoT (IIoT) [3], as well as the coexistence of heterogeneous networks [4], including both wired and wireless ones, in the same production line, are a direct proof of this evolution.

In this scenario, configuration, operation, management, and maintenance of both new (greenfield) and old (brownfield) machinery [5] are becoming increasingly complex tasks. On the other hand, these essential activities must be kept as simple as possible, is such a way to be easily accomplished by workers in the industry. Concerning equipment involved in networked systems, both wired (e.g., TSN [6]) and wireless communication technologies are often characterized by tight constraints, including real-time [7], reliability [8], [9], safety [10], security [11], [12], and power consumption [13], [14]. Therefore, proper configuration is demanded to maximize performance. One of the most complex parts related to the management of these kinds of apparatus is related to their diagnosis and maintenance in the case of malfunction [15].

Many recent proposals available in the scientific literature rely on the use of portable smart devices [16], and in some case augmented reality [17], to guide the worker throughout the steps involved in maintenance operations, which must be correctly executed also by non-perfectly trained personnel. Setting up an interactive process between the worker and an application executed on a portable device aimed, e.g., at identifying and solving a problem that has arisen in some part of the system, is an effective way to ease the implementation of the above actions, which in the most general case include configuration, operation, management, and maintenance.

Such activities typically require that the portable device is connected to either a server deployed on a local network or directly to the Internet, to retrieve all the required information or whatever is needed. Sometimes, information can even be stored in the device itself, but doing so limits flexibility tangibly, because one has to know in advance the kind of equipment involved and the related problems.

Unfortunately, in many cases the portable devices exploited to perform these operations do not have the possibility to use a communication network. This may be due to specific security policies within the company, which prevent access to the local network, or more commonly to the fact that the machinery is located in some place where there is no Internet access. Examples are installations in desert areas (high mountains, deserts, forests, etc.), petrochemical factories that, due to their extension, are sometimes not completely covered by a wireless network, railway exchanges, electrical substations, and so on.

The solution we are proposing in this work is to embed a programming language, we named QRscript, directly into a QR code, which is attached to the related part of the equipment. The worker can then scan the QR code with its portable device and execute the embedded program. In this way, the worker is enabled to interact with the program encoded in the QR code, which guides her/him toward the solution of a specific problem. To the best of our knowledge, this solution has not been proposed before, and this is the first time a QR code is used to store executable code.

It is worth pointing out that the application contexts where this technology can be profitably exploited are quite wide, and fall well outside the boundaries of factory automation. For example, it can be adopted in mountain trails to suggest possible routes with their respective characteristics, or to guide a user to correctly managing an emergency medical device, or finally to facilitate the use of devices of any kind

in developing countries[1], by providing a certain degree of interactivity without the need for a network connection.

In this work, in addition to presenting the main features of the QRscript programming language and its inherent ability to be extended, we also propose a specific sub-language (that is, a dialect) that permits to encode a decision tree within a QR code. All the steps involved in translating this dialect from a high-level programming language to a binary representation and generating the related QR code were analyzed in depth. Moreover, the opposite direction of the translation scheme, which goes from the QR code to the execution of the encoded program, has been also described and commented. Descriptions have been accompanied by a simple yet concrete example, to make it easier to understand all the involved steps.

This paper has the following structure: in Section II QR codes are briefly described, while the QRscript programming language and the main steps of the translation process (i.e., generation of the QR code and its execution) are defined in Section III. The specific dialect for encoding decision trees and the related example are thoroughly detailed in Section IV. Finally, some conclusive remarks are reported in Section V.

## II. QR CODE TECHNOLOGY

The quick response code technology [18], commonly known as QR code, is a two-dimensional barcode that was invented in 1994 with the aim of tracking vehicles during the manufacturing process. Besides the speed of recognition, its main advantage is the larger storage capacity when compared to pre-existing one-dimensional barcodes. The most recent standard specification related to QR code can be found in [19].

Currently, QR codes are used to encode different kinds of information like text, URLs for automatically connecting to a web page, and information to join a Wi-Fi network. In addition, they enable different types of applications, for instance to manage security [20] or mobility [21], for authentication, for payments, for augmented reality [22], for marketing purposes [23], etc.

QR codes can store four types of data, namely, *numeric*, *alphanumeric*, *binary*, or *kanij*. The latter is specifically intended for encoding Japanese symbols, and it was included because this technology was firstly proposed by Denso Wave, a Japanese automotive company. Different data types correspond to different information that can be encoded (i.e., numeric, alphanumeric, etc.), and consequently their correct use may lead to a reduction (or, vice-versa, to an increase) of the amount of information that can be embedded in the QR code.

For the purposes of this work, we rely on the binary mode, which is typically used to encode 8-bit ASCII characters with the ISO 8859-1 format. In our case, the code included in the QR code is the result of a compilation process, and hence it cannot be described efficiently using the ASCII coding scheme. Conversely, it is just a sequence of bits that represents the list of instructions of the program to be executed, which is suitably encoded in a specific binary format with the rules explained below.

[1]For example, in the Suzana village in Guinea-Bissau, many residents have mobile phones, but the Internet connection is available only in some areas and for short periods of time near humanitarian associations or missions.

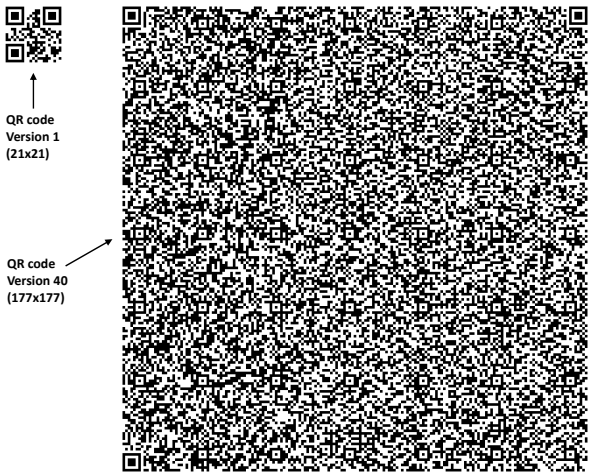



Fig. 1. Examples of QR code version 1 and version 40.

Different *versions* of QR codes are defined: the smallest one, in terms of the area and storing capacity is version 1, which is coded in a $21 \times 21$ matrix, while the largest is version 40, which is coded in a $177 \times 177$ matrix. Figure 1 shows two examples of QR codes, belonging to version 1 and version 40. In addition, there is the possibility to define different error correction levels, i.e., L (low), M (medium), Q (quartile), and H (high), which can be employed to counteract reading issues related, e.g., to damaged or dirty QR labels, hence increasing overall robustness. In the case of version 40 with a low level of error correction, the storage capacity is 2953 bytes.

The generation of compact QR codes and techniques to compress the information to be included were the subject of several scientific works. To improve storage capacity, colored QR codes [24], [25] were defined, as well as other techniques like multiplexing [26]. Due to the limited storage capacity of this technology, the ability to pack as much information as possible in a QR code is a primary requirement in most practical applications, and our proposal for embedding an executable program makes no exception. For this reason great attention was spent in the translation process for generating an extremely compact binary code.

## III. QRSCRIPT PROGRAMMING LANGUAGE

The QRscript programming language is an *interpreted* language that is defined inside and represented by a QR code. Figure 2 highlights all the steps involved in the *generation* of the QR code and in the *execution* of the embedded program. The latter step starts with the QR code being read (scanned) by the user and ends with its execution in a specific virtual machine. Here, the virtual machine has to be intended as a software environment that executes the binary code retrieved from the QR code. From a conceptual point of view, its

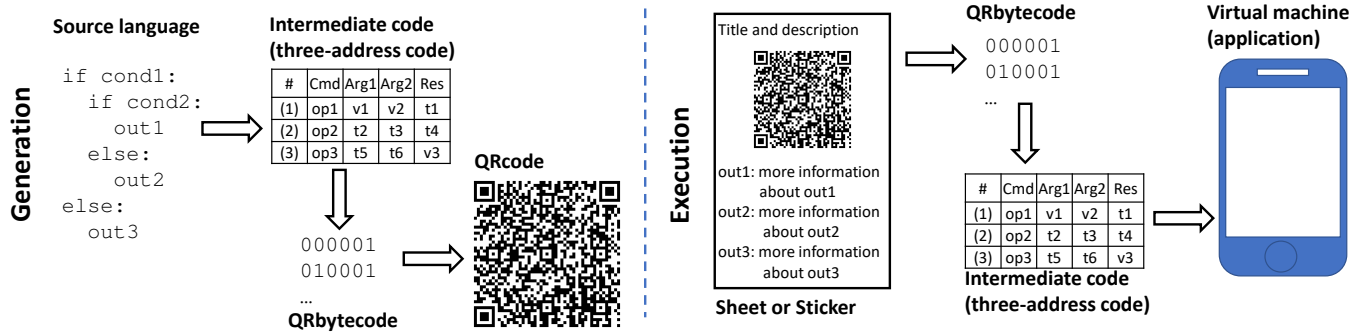



Fig. 2. Example of all the steps involved in the generation of the QR code (left side) and in the execution of the program from the QR code (right side).

purpose and behavior are similar to a Java Virtual Machine (JVM) for `java` or a `python` interpreter (which is actually a virtual machine).

### A. *QRbytecode generation*

The program to be encoded in the QR code can be described by means of a high-level programming language. The target of this first step is the generation of a sort of *bytecode*, we named *QRbytecode*, which represents the sequence of bits that must be encoded in the QR code in a binary form. The reason why we defined it as a bytecode is because the program is not intended to be executed by a dedicated hardware platform, but rather it is interpreted by a specific virtual machine. The main target of the QRscript programming language is the definition of the QRbytecode, which has stringent requirements about the dimension of the generated code, because it has to be embedded in a QR code whose maximum size is currently 2953 bytes (when version 40 with low error correction is employed).

The high-level source programming language is not the focus of this paper, and can be any language translatable into QRbytecode. Due to the aforementioned constraints related to code dimension and to satisfy different programming needs, a number of *dialects* (e.g., sub-languages) can be defined, each one with different features and expressive power. By limiting the kinds of operations that can be performed, some dialects may allow the generation of a more compact QRbytecode. At the same time, doing so constrains the characteristics and constructs of the high-level programming language.

The simplest dialect, which is the one we describe in this work, only supports decision trees. In this case, the source language could be limited to input operations, output operations, and "`if`" statements. Consequently, only the sequence and selection (choice) constructs of structured programming languages can be implemented, but not the repetition (loop). In addition, the ability to define and use variables could be omitted in some specific dialects. This implies that for these dialects the source programming language is necessarily streamlined, and only a small subset of algorithms can be actually implemented (which, however, includes many of those we are actually interested in).

As said before, the exact definition of the high-level programming language is outside the scope of QRscript, therefore all the pieces of code written in this language that appear in this work must be considered only as examples of a possible source language. When performing the translation from the source programming language to the QRbytecode, an intermediate representation can be possibly employed, like the *three-address code*. Typically, doing so eases both the design and the implementation of the translator. For this reason, in this work it was decided to follow this direction.

Once the QRbytecode has been generated and possibly optimized, it can be encoded in a QR code and, eventually, printed on a sheet or sticker that can be attached to the related industrial machinery, in a convenient place to be seen and used by workers. In addition to the QR code itself, the sticker can also contain some printed information written in natural language, which explains how to use the QR code and how to interpret the output of the program.

### B. *QRbytecode execution*

The execution process starts when the QR code is read by a client device (a smartphone, a tablet, or more in general an application) that, as the first step, performs the conversion from the QR code back to the QRbytecode. Then, the interpreter transforms this language into a suitable internal representation (which is typically the tree-address code) and executes it by means of a dedicated virtual machine.

From a practical point of view, all the applications installed in the client need the ability to interpret and execute the code stored in the QRbytecode. Typically, it is an Android or Apple iOS app, which can be possibly customized for specific application contexts and modified by the system manager (or users, in general) so as to meet some requirements. In particular, the way input and output operations are performed is highly dependent on the characteristics of the industrial process and environment. For instance, in dusty/dirty industrial environments, input can be made easier if the application uses large fonts for the condition (i.e., when asking questions) and large on-screen buttons for every possible response. Likewise, in contexts where hands-free operation is demanded, the use of

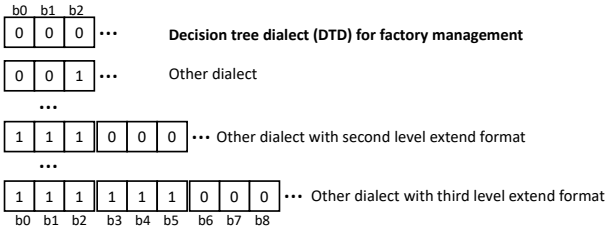



Fig. 3. Definition of the initial part of the QRbytecode specifying the dialect.

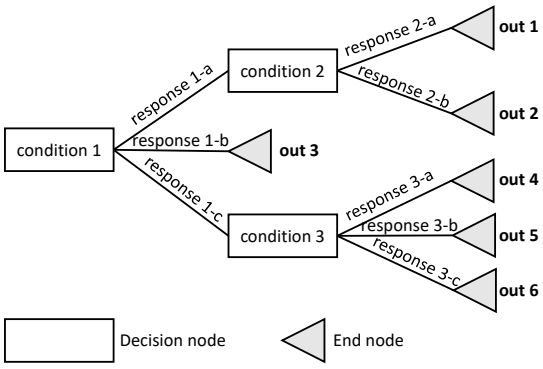



Fig. 4. Example of a decision tree without chance nodes.

text to speech (TTS) and automatic speech recognition (ASR) modules could facilitate the interaction with the worker.

It is worth stressing that QR codes embedding QRscript programs are mostly valuable when the device used to carry out configuration and guided diagnostics is not provided with a reliable/stable Internet connection, otherwise a simple web browser coupled with a QR code containing the URL of a suitable web resource is perhaps the best choice. This may also happen in those installations where wireless access is not authorized for devices of the maintenance staff.

### C. QRscript dialects

Different dialects can be defined for the QRscript programming language, in order to reach a compromise between the dimension of the generated QRbytecode and the expressiveness of the programming language. In the current proposal, the first 3 bits of the QRbytecode are used to identify the dialect that is being used in the following part of the code. As shown in the schema of Figure 3, the dialect characterized by code `000` corresponds to the decision tree dialect (DTD), which is the one analyzed and described in this work.

As a matter of fact, the operations that are requested for configuring and maintaining industrial equipment can be often described using a simplified version of a *decision tree* model, with only *decision* and *end* nodes, but without any *chance* nodes (chance nodes are a kind of nodes that represent a probabilistic decision). In the context of this work, we want that all the decisions are taken by the user who is interacting with the algorithm. An example of this kind of decision tree is reported in Figure 4, in which every decision node can have more than two responses (we decided not to limit it to binary decisions).

For every outcome, which is represented by an end node, a specific *decision rule* is associated. In the case of the decision tree of Figure 4, six decision rules can be generated. For the first outcome the decision rule is `if (condition 1 == response 1-a) AND (condition 2 == response 2-a) then out 1`. A decision tree can be directly (and easily) transformed into a *flow chart*, which is a more efficient representation because it does not require the use of variables to store the results of conditions, and is also more deterministic and faster from the point of view of the execution time.

The other combinations of the first 3 bits of the QRbytecode can be used to define other dialects. As previously mentioned, they are just languages with different capabilities in terms of the available instruction set and the dimension of the QRbytecode. As far as we know, this is the first time that a programming language is compiled and fit into a QR code. Nevertheless, in the case some pre-existing languages were defined, they could be easily included in this representation by assigning them a specific dialect code. The format for representing dialects is defined in such a way that it can be easily extended to a number of dialects that exceeds what can be encoded on three bits. In particular, if the first three bits are equal to `111`, the next 3 bits are used to identify additional dialects. Likewise, if the first six bits are all equal to `1`, the next 3 bits can be used to identify new dialects, and so on.

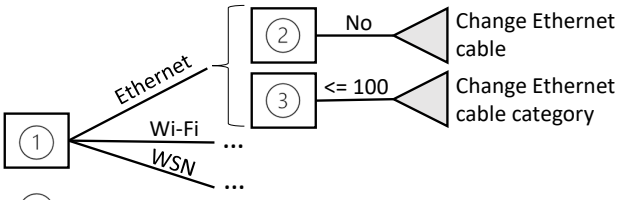
## IV. Decision tree dialect

The DTD is a dialect of the QRscript programming language that permits to encode well-known decision trees without chance nodes. Consequently, it can be exploited to define algorithms to help workers to solve certain problems, which are for instance related to network (or machinery) configuration and maintenance. In DTD, an improved version of decision tree is defined that supports the concatenation of several decision trees. In particular, every condition has a default response, we named *other*, that, if selected, permits the user to move to the following decision tree.

In this subsection, DTD is described and defined starting from a compact and illustrative example. Some of the main parts involved in the generation of the QRbytecode, starting from a sample high-level programming language, were implemented using the `jflex` scanner and the `cup` parser, which produce a translator implemented in the `java` language. However, they can be implemented with any other bottom-up parser, like the pair `flex/bison` that produces a translator in the `C` programming language, or `ply` that produces a translator in the `Python` programming language.

To make understanding easier, we explained this dialect with an example in a top-down fashion, i.e., starting from a high-level graphical representation to arrive to the binary QRbytecode that is encoded in the QR code.

**Graphical representation of a decision tree**




① Which kind of technology has communication problems?
② Is link status active?
③ What is the speed in Mbps?

**Corresponding high-level language representation**

```
input 1
# input "Which kind of technology has communication
         problems?"
if "Ethernet":
  input 2   # input "Is link status active?"
  if "No":
    print "Change Ethernet cable"
    exit
  inputs 3  #inputs "What is the speed in Mbps?"
  if <= 100:
    print "Change Ethernet cable category"
    exit
  exit
else if "Wi-Fi":
  ...
else if "WSN":
  ...
# No solution
```

Fig. 5. Example of decision tree and corresponding high-level language that can be encoded in a QR code using QRscript ("#" denotes comments).

### A. High-level representation

Figure 5 shows the graphical representation of a decision tree that can be implemented using QRscript, and the corresponding representation through a Python-like pseudo-code. In this simplified example, the decision tree is used to guide the worker in solving a connectivity problem. After asking the user to identify the kind of technology that is having a problem (i.e., Ethernet, Wi-Fi, or WSN), in the case the response is "Ethernet" the algorithm described by the decision tree tries to identify the problem by asking if the link status is active. Typically, if the Ethernet cable is disconnected or broken the link status led is inactive. If the user selects a response other than "No", the algorithm switches to the next concatenated decision tree, identified with the number ③ as root, which corresponds to the question "What is the speed in Mbps?". If the value entered by the user does not match the condition "`<= 100`" the algorithm ends without a solution, otherwise it recommends to "Change Ethernet cable category" (we assume that cabling must support Gigabit Ethernet, whose expected speed is $1000\,\mathrm{Mb/s}$).

The representation of the algorithm with a high-level programming language is reported in the lower part of the same

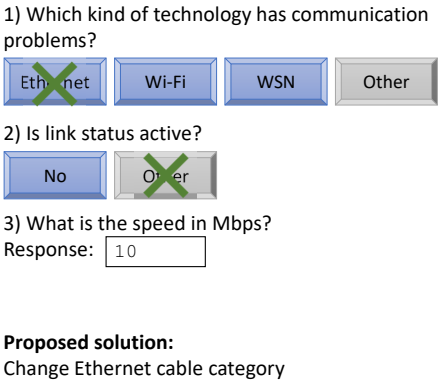



Fig. 6. Example of interaction between the user and the QR code mediated by a suitable application provided with a graphical interface (the interface can be automatically built when executing the QRbytecode of the DTD).

figure. The `input/inputs` instruction prints a message on the screen and requests a string as input, which can be entered by the user either *directly* through a textual interface, when the `inputs` instruction is used, or *indirectly*, for instance by means of a decision button, in the case of the `input` instruction. In this dialect only the `string` type can be used for input values, and the virtual machine aimed at executing the code tracks only the last entered value (i.e., every time a new `input` instruction is executed the previously entered string is definitely lost).

When the argument of the instruction is an unsigned integer, as for `input 1` and `inputs 1`, a reference is printed instead of a string. See, for instance, the symbol ①. The idea behind this option is that the string associated to reference ① can be physically printed on a sheet or a sticker (likely the same on which the QR code is found), instead of being encoded in QRbytecode. In this way, a fair amount of space is saved in the QR code, since the characters that make up the string are left out of it.

Comparison with integer (e.g., `<= 100`) and floating point (e.g., `> 3.5`) values is also possible. In these cases, the virtual machine performs an automatic conversion of the string entered by the user into an internal integer or floating point value, respectively.

A possible interaction between a user and the application (i.e., the virtual machine that executes the QR code), based on the above discussed program, is depicted in Figure 6. As shown on the right side of Figure 2, the application and its virtual machine are the final step in the chain related to code execution, which is directly based on the QRbytecode automatically generated by the translator and read from the QR code. We decided to start our discussion about the whole chain from the end, and to begin with the interaction with the user, to permit a complete and thorough understanding of the example. As can be seen, the questions asked by the algorithm depend on the previous responses provided by the user. To each question, a default response named "Other" (and reported in grey in the figure) is added. If selected, this response

```
(1)  input "Which kind of technology has
               communication problems?"
(2)  if "Ethernet" (6)
(3)  if "Wi-Fi" (15)
(4)  if "WSN" (20)
(5)  goto (25)
(6)  input "Is link status active?"
(7)  if "No" (9)
(8)  goto (10)
(9)  printex "Change Ethernet cable"
(10) inputs "What is the speed in Mbps?"
(11) ifc <= 100 (13)
(12) goto (14)
(13) printex "Change Ethernet cable category"
(14) printex ""
(15) # Code related to Wi-Fi
     ...
(20) # Code related to WSN
     ...
(25) printex ""
```

Fig. 7. Example of three-address code derived from the high-level language representation reported in Figure 5.

causes the program flow to jump to the next concatenated question (or decision tree). In the case of input sets made up of enumerated values (e.g., related to an `input` instruction), they could be represented by a number of buttons. Instead, a typical rendering for an `inputs` instruction is through a textual box.

### *B. Translation of DTD to three-address code*

Starting from a high-level description of the decision tree (graphical or textual), an intermediate representation based on the three-address code can be automatically obtained by means of a translator. It is worth remarking again that the high-level language proposed in this work is just an example. The specifications related to QRscript in general, and DTD in particular, are related to QRbytecode generation, and from a certain perspective to three-address code generation as well, because there is a direct mapping between these two formalisms.

Three-address code typically consists of a numbered/ordered list of quadruples, where the fields of each quadruple represent the instruction (or operation), its first and second arguments, and the result, respectively. For some instructions the second argument is not used.

In DTD three-address code, seven instructions are defined:

- `input <constant>` (or `inputs <constant>`): requires an indirect (or direct) input of a string, respectively. The term `<constant>` can be either a `<string>` or a `<reference>` (i.e., an unsigned integer value).
- `print <constant>`: prints the string or the reference identified by `<constant>`.
- `printex <constant>`: same as `print`, but after the instruction the execution is terminated.
- `goto (x)`: jumps to instruction number `x`, which is identified by the label `(x)`.
- `if <string> (x)`: if the last entered string value is equal to `string`, it jumps to instruction number `x`.
- `ifc <rel_op> <operand> (x)`: if the comparison between the last entered string value and the `<operand>` using the `rel_op` relational operator is true, it jumps to instruction number `x`. If `<operand>` is an integer or a floating point value, the last entered string is automatically converted to an integer or a float number, respectively. An error is returned if the conversion is not possible. Possible `rel_op` are `==`, `!=`, `<=`, `>=`, `<`, and `>`. This is the only instruction that makes use of both the arguments of the three-address code representation based on quadruples.

The most important and complex part to pay attention in three-address code generation is related to semantic actions aimed to the generation of unconditional (`goto` instruction) and conditional (`if` and `ifc` instructions) jumps. In any case, this aspect is completely addressed by compilers' theory, and for bottom-up parsers it can be easily solved through the use of specific inherited attributes.

Figure 7 reports the three-address code obtained from the decision tree described by the high-level language in Figure 5. The fields of each quadruple are separated by means of white spaces. Analyzing one of the possible execution flows, if the answer to the first choice about the network technology is "Ethernet" a jump is performed to instruction `(6)`, in which the second input is requested. If the user, at this point, replies "No", there is a jump to instruction `(9)`. Then, the proposed solution is consequently displayed ("Change Ethernet cable"), after which the program terminates. In the case the reply to the second question in line `(6)` is "Other" (whose meaning is that link status is active), a jump is made to instruction `(10)`, and another input is requested, which in this case is direct. If the user enters a value greater than 100, the `goto (14)` instruction is executed. In this case the program jumps to line `(14)` and, with the instruction `printex ""`, it terminates its execution without printing any message.

### *C. QRbytecode generation for DTD*

Starting from the three-address code representation, the QRbytecode of the DTD can be directly generated. In this step, the main goal we pursued was to reduce the dimension of the generated code as much as possible. Consequently, when defining the translation rules, we paid great attention to generating a compact code. This section is aimed at the definition of the syntax (and semantics) of the QRbytecode, and it shows how the DTD dialect can be transformed bitwise into bytecode and how, starting from bytecode, the relevant three-address code can be retrieved. The description on how to implement the virtual machine that executes the QRbytecode has not been reported because it does not add any relevant details.

The first three bits of the QRbytecode are `000`, to identify the DTD dialect and to instruct the virtual machine to use DTD rules. Each instruction in three-address code has a direct mapping in QRbytecode. In particular, the first three bits (i.e., the instruction *code*) identify the instruction: `000` for `input`, `001` for `inputs`, `010` for `print`, `011` for `printex`, `100`

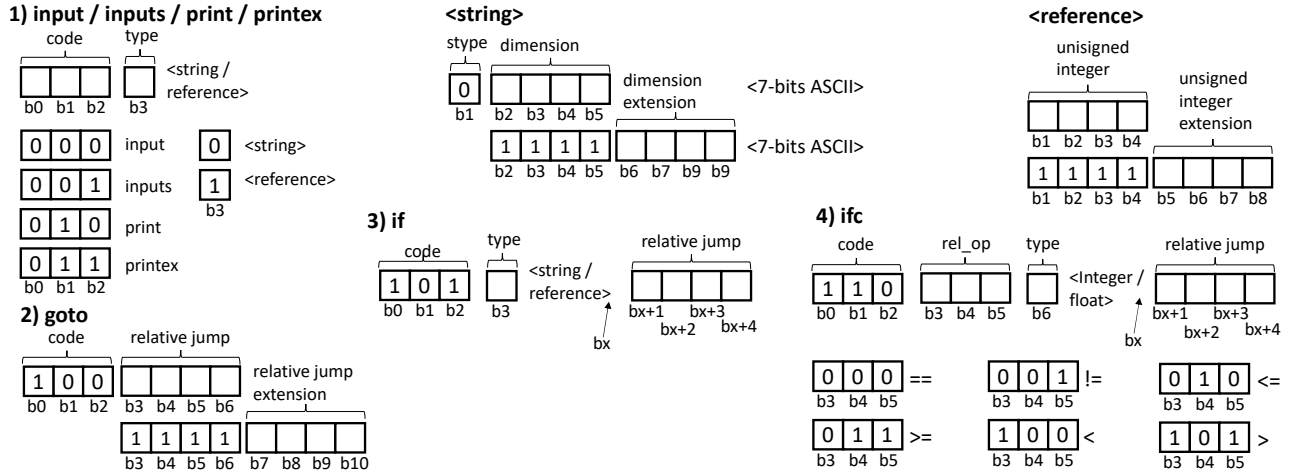



Fig. 8. Conversion rules from three-address code to QRbytecode.

for `goto`, `101` for `if`, and `110` for `ifc`. The code `111` was left unused to permit possible extension of the instruction set. A schematic view of how the instructions have been encoded is reported in Figure 8.

Starting from the input/output instructions (i.e., `input`, `inputs`, `print`, and `printex`), the *type* bit after the code identifies whether the provided output is either a string (`<string>`) or a reference (`<reference>`).

Since storing strings requires a fair amount of space, we paid attention to their definition. Each string in its binary representation starts with a *stype* bit which, if set to `0`, defines a compact string, and it is followed by a *dimension*, encoded as an unsigned integer on `4` bits, which represents the number of characters that are subsequently encoded. Similarly to the encoding of dialects, when the previous `4` bits are equal to `1111`, the dimension is extended recursively by another `4` bits. Characters are encoded using a 7-bit character set, typically ASCII, but if both the generation and the execution steps agree on the standard to be used, other solutions like ISO/IEC 646 and 8-bit character sets are other viable options. Clearly, the latter increases the dimension of the generate binary code. The value *stype=1* was currently left unused (reserved), and can be exploited to extend string coding to other character sets. References are coded as unsigned integers on `4` bits, and can be recursively extended to represent numbers of any size.

The `goto` instruction executes unconditional jumps, and is identified by the starting sequence of bits `100`. The *code* is followed by a *relative jump*, which specifies a recursively extensible jump width, encoded as an unsigned integer on `4` bits, that represents the number of instructions to be skipped starting from the next instruction (i.e., `0000` means jump to the next instruction). It is worth pointing out that only forward jumps are possible, because DTD does not foresee loops. Conversely, in the case of a dialect with backward jumps, the relative jump has to be encoded as a signed integer.

Two conditional jump instructions are defined, namely `if` and `ifc`. The first is identified by the code `101`, followed by a type that specifies whether the value inserted by the user has to be compared with either a string or a reference, and a relative jump is executed when the comparison matches. The `ifc` instruction differs from `if` because, after the code `110`, it defines a relation operator (*rel_op)* using a 3 bits representation. For this instruction, *type=0* identifies a signed integer number, while *type=1* identifies real number encoded as a half-precision floating point on 16 bits. Integer numbers are stored using the *two's complement* representation with 16 and 32 bits. The first bit is used to select between 16 and 32 bits representations.

At the end of the translation process, the QRbytecode is padded with up to 7 bits taken from the sequence `1000000`, in such a way that the final dimension of the generated QRbytecode is a multiple of `8` bits, as possibly requested by the *binary* coding mode. If the number of bits added by padding is not enough to encode a complete instruction, the virtual machine will simply discard it at runtime. Otherwise, the instruction the virtual machine will interpret in the padding corresponds to a `goto` conditional jump to the next instruction (that does not exists). In both cases no problems are caused to the virtual machine.

Using the translation rules described above, the three-address code presented in Figure 7 can be directly translated to QRbytecode, and vice-versa. In terms of storage occupation, the space needed to store instructions from (1) to (14) of Figure 7 is `654` bits (82 bytes with padding). Instead, the translation between QRbytecode and QR code is trivial, because a number of libraries exist free of charge to perform this operation[2].

## V. Conclusions

The ability to embed an executable program inside a QR code is of particular interest, because it enables a number of applications. Decision support and configuration, as well as maintenance and diagnostics of industrial equipment and

[2]An example is the `PyQRCode` module for the `Python` programming language.

networks, are just few examples of the possible scenarios that can take advantage by this technology. Systems located far away from the main industrial plant, where Internet access is not available (or unreliable) for whatever reason, including military applications, can all benefit from our technology. Unlike techniques that make the device used for configuration (a mobile phone) interact directly with intelligent equipment by means of, e.g., NFC or visible light communication, our solution does not require any modification to the existing equipment and is extremely inexpensive.

The proposed QRscript programming language has been conceived bearing extensibility in mind, and permits to define a number of sub-languages termed dialects. In this work, all the steps needed to generate a QR code containing a program in binary form starting from its description given in a high-level programming language, and the corresponding chain with which the program is read from the QR code and executed on a virtual machine, have been thoroughly detailed. The DTD dialect, which is aimed at coding a decision tree inside a QR code, was carefully defined, along with an intermediate representation of the program based on three-address code. The description of the dialect and the translation process were explained by using a very simple, yet realistic example.

We believe that the ability to embed a program in a QR code that can be read and executed on handheld/portable devices is a really appealing option. Consequently, our future works on this subject will focus on identifying new areas where this technology can be profitably employed, and on providing concrete examples to show how this can be done. Moreover, we plan to define additional dialects in order to enhance the features and capabilities of QRscript, and to add security features based on asymmetric cryptography to distinguish among user roles (those enabled to forge QR codes, to just execute them, or to do nothing).

## REFERENCES


[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128610001568

[2] H. Cañas, J. Mula, M. Díaz-Madroñero, and F. Campuzano-Bolarín, "Implementing Industry 4.0 principles," *Computers and Industrial Engineering*, vol. 158, p. 107379, 2021.

[3] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial Internet of Things: Challenges, Opportunities, and Directions," *IEEE Trans. Ind. Informat.*, vol. 14, no. 11, pp. 4724–4734, 2018.

[4] S. Scanzio, L. Wisniewski, and P. Gaj, "Heterogeneous and dependable networks in industry – A survey," *Computers in Industry*, vol. 125, p. 103388, 2021.

[5] S. K. Panda, L. Wisniewski, M. Ehrlich, M. Majumder, and J. Jasperneite, "Plug & Play Retrofitting Approach for Data Integration to the Cloud," in *16th IEEE International Conference on Factory Communication Systems (WFCS 2020)*, 2020, pp. 1–8.

[6] B. Caruso, L. Leonardi, L. L. Bello, and G. Patti, "Design of a framework for enabling TSN support in heterogeneous platforms with virtualization and preliminary experimental results," in *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021)*, 2021, pp. 01–04.

[7] G. Cena, S. Scanzio, and A. Valenzano, "Experimental Evaluation of Techniques to Lower Spectrum Consumption in Wi-Red," *IEEE Transactions on Wireless Communications*, vol. 18, no. 2, pp. 824–837, 2019.

[8] S. Scanzio, M. G. Vakili, G. Cena, C. G. Demartini, B. Montrucchio, A. Valenzano, and C. Zunino, "Wireless Sensor Networks and TSCH: A Compromise Between Reliability, Power Consumption, and Latency," *IEEE Access*, vol. 8, pp. 167 042–167 058, 2020.

[9] G. Cena, S. Scanzio, and A. Valenzano, "Improving Effectiveness of Seamless Redundancy in Real Industrial Wi-Fi Networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2095–2107, 2018.

[10] T. R. Doebbert, C. Cammin, and G. Scholl, "Safety Architecture Proposal for Low-Latency Sensor/Actuator Networks using IO-Link Wireless," *IEEE Access*, vol. 10, pp. 3030–3044, 2022.

[11] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano, "A comprehensive approach to the automatic refinement and verification of access control policies," *Computers and Security*, vol. 80, pp. 186–199, 2019.

[12] M. Ehrlich, L. Wisniewski, H. Trsek, D. Mahrenholz, and J. Jasperneite, "Automatic mapping of cyber security requirements to support network slicing in software-defined networks," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2017)*, 2017, pp. 1–4.

[13] S. Scanzio, G. Cena, A. Valenzano, and C. Zunino, "Energy Saving in TSCH Networks by Means of Proactive Reduction of Idle Listening," in *Ad-Hoc, Mobile, and Wireless Networks*, L. A. Grieco, G. Boggia, G. Piro, Y. Jararweh, and C. Campolo, Eds. Cham: Springer International Publishing, 2020, pp. 131–144.

[14] G. Cena, S. Scanzio, and A. Valenzano, "Ultra-Low Power Wireless Sensor Networks Based on Time Slotted Channel Hopping with Probabilistic Blacklisting," *Electronics*, vol. 11, no. 3, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/3/304

[15] A. Muller, A. Crespo Marquez, and B. Iung, "On the concept of e-maintenance: Review and current research," *Reliability Engineering and System Safety*, vol. 93, no. 8, pp. 1165–1187, 2008.

[16] E. Permin, F. Lindner, K. Kostyszyn, D. Grunert, K. Lossie, R. Schmitt, and M. Plutz, *Smart Devices in Production System Maintenance*. Cham: Springer International Publishing, 2019, pp. 25–51.

[17] C. Toro, C. Sanín, J. Vaquero, J. Posada, and E. Szczerbicki, "Knowledge Based Industrial Maintenance Using Portable Devices and Augmented Reality," in *Knowledge-Based Intelligent Information and Engineering Systems*, B. Apolloni, R. J. Howlett, and L. Jain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 295–302.

[18] S. Tiwari, "An Introduction to QR Code Technology," in *International Conference on Information Technology (ICIT 2016)*, 2016, pp. 39–44.

[19] ISO/IEC, "Information technology - Automatic identification and data capture techniques - QR Code bar code symbology specification," in *ISO/IEC 18004:2015*, 2015, pp. 1–117.

[20] K. Saranya, R. Reminaa, and S. Subhitsha, "Modern applications of QR-Code for security," in *IEEE International Conference on Engineering and Technology (ICETECH 2016)*, 2016, pp. 173–177.

[21] S. L. Fong, D. Wui Yung Chin, R. A. Abbas, A. Jamal, and F. Y. H. Ahmed, "Smart City Bus Application With QR Code: A Review," in *IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS 2019)*, 2019, pp. 34–39.

[22] T.-W. Kan, C.-H. Teng, and W.-S. Chou, "Applying QR Code in Augmented Reality Applications," in *Proceedings of the 8th International Conference on Virtual Reality Continuum and Its Applications in Industry*, ser. VRCAI '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 253–257. [Online]. Available: https://doi.org/10.1145/1670252.1670305

[23] C. Teuta, S. P. Payal, A. Ramesh, and T. Sakaguchi, "QR Code: A New Opportunity for Effective Mobile Marketing," *Journal of Mobile Technologies, Knowledge and Society*, vol. 2013, p. ID748267, 2013.

[24] A. Abas, Y. Yusof, R. Din, F. Azali, and B. Osman, "Increasing data storage of coloured QR code using compress, multiplexing and multilayered technique," *Bulletin of Electrical Engineering and Informatics*, vol. 9, no. 6, pp. 2555–2561, 2020. [Online]. Available: https://beei.org/index.php/EEI/article/view/2481

[25] M. Arora, C. kumar, and A. K. Verma, "Increase Capacity of QR Code Using Compression Technique," in *3rd International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE 2018)*, 2018, pp. 1–5.

[26] S. Vongpradhip, "Use multiplexing to increase information in QR code," in *8th International Conference on Computer Science Education*, 2013, pp. 361–364.