Informatics in Education, 2020, Vol. 19, No. 2, 223–238

DOI: 10.15388/infedu.2020.11

Scratch-Based User-Friendly Requirements Definition for Formal Verification of Control Systems

Iwona GROBELNA
Faculty of Computer, Electrical and Control Engineering, University of Zielona Góra
Zielona Góra, Poland
e-mail: i.grobelna@iee.uz.zgora.pl

Received: September 2019

Abstract. Control systems are becoming ever more commonly used in everyday life. This is true both in industry and in the domestic domain, in the form of e.g., smart home systems. The quality of such systems can be increased by using formal verification methods, such as the model checking technique, to make sure that the designed system fulfills all user requirements. The requirements are usually written as temporal logic formulas. However, the technical skills of future users or the mathematical background knowledge of the developers are not always sufficient to support the essential stage of verification. In the paper we propose to use the Scratch-based user-friendly approach to define our own scenarios for a control system, in order to avoid focusing on the mathematical notation of temporal requirements. The specified properties can then be transformed into temporal logic formulas and used directly in the model checking process. Hence, the verification phase is simplified and more team members can participate in the engineering of requirements. An empirical study with students has shown that the proposed approach can be used in practice.

Keywords: control systems, formal verification, logic controller, model checking, requirements engineering, specification.

1. Introduction

The modern world is being expanded by technological evolution. More and more software and hardware systems are being used in everyday life. Control systems are becoming more accessible to ordinary citizens, e.g., smart home systems. Smart homes (Alam *et al.*, 2012; Bitterman and Shach-Pinsly, 2015; Klimek and Rogus, 2015) are required to be user-friendly and offer modern environments. The automation control systems for smart homes are purchased by customers who are not always very familiar with the technology itself, and therefore such systems should be easy to manage and use. Future users should also be able to specify how their automation system should work.

On the other hand, formal verification allows for an increase in the quality of any hardware or software system (Woodcock *et al.*, 2009; Kropf, 1999), considering also real-life processes (Grobelna *et al.*, 2017) or exchangeable modules in reconfigurable control systems (Wisniewski and Grobelna, 2018; Grobelna, 2018; Wisniewski, 2018). However, one should remain aware of the limitations of formal methods (Tempelmeier, 2011), including environmental influences, unaccountable world knowledge or the misbehavior of neighbouring systems. One of the most approved methods is the model checking technique (Pakonen *et al.*, 2016; Clarke *et al.*, 1999l Emerson, 2008; Jiang *et al.*, 2018), which has recently been entering slowly even into quantum systems (Ying and Feng, 2018). Generally speaking, in model checking the specification is validated against some user-defined requirements. If any of them cannot be satisfied, appropriate counterexamples are generated. It is important that the list of requirements is as complete and unambiguous as possible, since only the specified properties can be checked. Hence, the quality of formal verification depends on the quality of requirements (Bozzano *et al.*, 2014). Additionally, the requirements are needed to make sure that the system is designed properly, otherwise a final product can be released that does not meet customer wishes (Robertson and Robertson, 2012).

For the model checking process, requirements are expressed as temporal logic formulas (Huth and Ryan, 2004; Ölveczky, 2017). However, the mathematical approach is not very convenient to use, especially for non-engineers, and the effort to formalize the functional requirements is the foremost challenge (Pakonen *et al.*, 2016). Moreover, user awareness of the safety issues is not always high enough (Grobelna *et al.*, 2018), so the tasks performed by the user should be either automated (if possible) or simplified, so that they are neither time-consuming nor problematic in use.

During the last few decades, the topics of formal verification and model checking have been part of the ongoing research. However, to the best of the author's knowledge, none of the existing methods of requirements specification specifically deal with control systems, and at the same time are simple to use for the common user not familiar with temporal logic theory.

All these facts, combined with detailed analysis of the state of the art, have built a strong motivation to consider a user-friendly form of requirements definition for control systems, namely, based on Scratch. Scratch (Resnick *et al.*, 2009; Scratch Project Editor) is a project at the MIT Media Lab, provided free of charge, and targeted especially at young people for programming of interactive stories, games and animations. It is used in more than 150 different countries and available in more than 40 languages, which makes it very popular in many cultures. The simplicity of rules makes it also suitable for adults or elderly people not so familiar with information technology.

Therefore, a novel method to define the control system requirements is proposed in this paper, dedicated especially to non-engineers, to designers of nontechnical user experience and to ordinary future end-users of such control systems. It may be also well suited for learning temporal logic among students, which fact has been proved in an empirical study. The Scratch-based specification is quite similar to the natural language, with the difference being that it follows some rules and only the defined blocks of words can be used. It should be highlighted that the proposed approach is a simple

and user-friendly method of requirements definition for control systems, supporting formal verification (model checking) of the specification.

The paper is organized as follows. Section 2 presents the state of the art regarding methods of requirements specification in terms of their simplicity of use and the benefits of using the Scratch environment. Section 3 contains the main contributions of this paper and introduces the novel method of Scratch-based user-friendly definition of temporal logic requirements for control systems. Section 4 summarizes some primary empirical studies among students. Finally, Section 5 concludes the paper.

2. State of the Art

In order to perform the model checking process, requirements of a control system have to be written as temporal logic formulas. They can be firstly written in natural language and then translated into temporal logic. However, the translation process is error-prone to ambiguities related, e.g., to diverse terms or complex sentences, and to the copy-paste operation of written sentences (an empirical study on different programming languages shows that it is a common human behavior to copy-paste some data (Ahmed *et al.*, 2015)). An approach based on natural language is presented in (Barza *et al.*, 2016), where the Controlled Natural Language (CNL, being a subset of English that obeys a formal grammar) is proposed as a language for writing requirements. Additionally, a translator from a CNL to the modelling language of the NuSMV model checker has been implemented. Another interesting requirements specification technique (for software), called seam-less requirements, is proposed in (Naumchev and Meyer, 2017), where natural language is used together with formal components. Unfortunately, it is only applicable to non-concurrent programs.

The efficient and user-friendly solution is through the use of UML activity diagrams (one of the most common diagrams of the Unified Modelling Language (OMG Unified Modelling Language)), proposed in (Grobelna and Grobelny, 2015), where only the basic technical knowledge (but still) of the UML language is needed. A quality assured model-driven requirements engineering and software development method is proposed in (Lengyel *et al.*, 2015), based on the modelling of software requirements oriented towards an automatic generation of several artifacts.

More research on user-friendly specification methods for requirements, including, e.g., property patterns, visual languages (such as Graphical Interval Logic, Property Sequence Charts or Live Sequence Charts) and Property Specification Language, being an IEEE standard, can be found in the survey paper (Pakonen *et al.*, 2016), where the conclusion is given that finding a user-friendly specification language with the right balance between expressiveness and readability is still an up-to-date and important research topic. In (Dillon *et al.*, 1994; Smith *et al.*, 2001; Vyatkin and Bouzon, 2008) the requirements are presented on timing diagrams, which – due to their technical nature – can sometimes be relatively difficult to use. Property Sequence Charts (Autili *et al.*, 2007) as well as Live Sequence Charts (Brill *et al.*, 2004) use UML diagrams to define scenarios. Simple graphical scenarios (for the software domain) are also proposed in (Asteasuain

and Braberman, 2017), where a declarative language is introduced. In (Beckert *et al.*, 2017) Generalised Test Tables are presented as a specification language for reactive systems, which are then encoded into verification conditions. However, all these approaches are not very close to the natural language nor do they seem to be very easy to use for non-engineers in practice.

On the other hand, an approach to visualize logical formulas is proposed in (Klimek, 2018), where a widely accessible system in the form of a web application is initially implemented and the visualization methods are based on graphs. It does not refer directly to requirements definition, but the aim of the tool is to help to understand, analyze and examine logical formulas, what is also a significant step towards the user. There are also many publications regarding requirements prioritizations, such as the MoSCoW technique (Tudor and Walter, 2006) or the Hundred Dollar Method (Davies, 2005), more information can be found in a comparison of the commonly used techniques in (Hudaib *et al.*, 2018).

Other research shows that Scratch is used with success in various projects. It can be used to code a user's own interactive stories, animations or games. It teaches students to think creatively, reason systematically and work collaboratively. The online editor is easy to use, the elements just need to be drag-and-dropped from the list into the main coding area and joined with the other elements. Each type of element has its own colour and some of them can be customized by specifying their characteristic. The literature overview reveals that Scratch is a good option for teaching students some new concepts. They can learn basic code structures and algorithmic thinking by using simple elements, and the Graphical User Interface is intuitive and easy to use. Results of an experiment conducted in a primary school (Gülbahar and Kalelioğlu, 2014) have shown that all the students liked programming and wanted to improve their programming skills after the Scratch course. Scratch is also commonly used at universities as an introduction to programming course for engineering students (Ozoran *et al.*, 2012), making programming more enjoyable and more visual. Using the Scratch environment for learning programming highly motivates students and empowers them to pursue their studies in programming (Ouahbi *et al.*, 2015). It is also well suited to helping understand the elements of logic and mathematics (Sáez-López *et al.*, 2016) or to being used in an integrated approach for enhancing the learning of computer programming (Cárdenas-Cobo *et al.*, 2018). There are also approaches in other domains of computer science not related directly to education, such as (Nergaard *et al.*, 2015), where a Scratch-based policy editor for the extensible Access Control Markup Language (XACML) is proposed, which just simplifies building the XACML policies. The disadvantage of Scratch may be bad programming habits, such as the repetition of code and object naming (Moreno and Robles, 2014), but the advantages still outweigh the disadvantages.

Pattern-based analysis of automated production systems, making formal verification accessible to engineers, is proposed in (Campos and Machado, 2009). The approach is a step towards the user, here a specialist already working in the industry. In the paper, we are going to an earlier stage of engineer career, namely to academic education of future experts.

Taking into account that Scratch is often introduced at schools and universities to teach children and students how to code and to help understand some new knowledge, it can be deduced that adults too who are not familiar with information technology can also easily adjust to its rules and use its online editor. This is the key issue for the conducted research and for Scratch-based user-friendly definition of requirements proposed in this article – to make the requirements definition as simple as possible and accessible to a non-expert user.

3. Expressing Requirements Using Scratch-Based Approach

The user-friendly Scratch-based method of requirements definition is an important step in the design process of control systems (illustrated in Fig. 1) and fits perfectly into the design flows of logic controllers. The primary specification of a control process (e.g., in the form of control interpreted Petri nets or UML diagrams) is formally written as an abstract rule-based logical model (Grobelna *et al.*, 2017), suitable both for model checking and synthesis. To perform formal verification, a list of requirements to be checked is also needed. In the proposed approach, they are defined with Scratch and transformed into temporal logic formulas. If in the model checking process any of the properties are not satisfied, appropriate counterexamples are generated and both the specification and the requirements list should be revised. Necessary modifications have to be made and formal verification is then performed once more. After successful verification the design flow can proceed with logic synthesis and the system can be physically implemented using dedicated hardware platforms (e.g., FPGA devices).

3.1. *Basic Elements*

Requirements for control systems are usually divided into two types: regarding safety (i.e., something bad will never happen) and liveness (i.e., something good will eventually happen). The safety properties are especially important for the correct functionality of the designed system and they can be easily verified using the model checking tech-

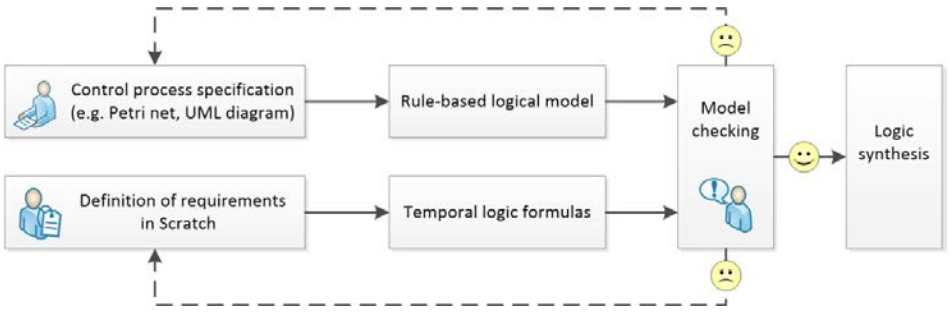


Fig. 1. Design flow with Scratch-based definition of requirements.

Table 1
Basic operators in the LTL temporal logic

Operator	Meaning	Example
G	globally / forever / always	*G p* formula *p* is true forever (that is in all states)
X	next	*X p* formula *p* is true in the next state
F	finally / sometimes	*F p* formula *p* is true finally (that is in some states)

nique. In the model checking process (we use the nuXmv tool (Cavada *et al.*, 2014)) the properties are expressed as LTL (linear-time temporal logic) or CTL (branching time temporal logic) formulas (Huth and Ryan, 2004). In the paper we focus on the LTL formulas, where single temporal operators are used: *G*, *X* and *F* (explained in Table 1), possibly combined with Boolean logic (and, or, not). The selection of temporal operators to be used in Scratch was based on the statistical usage of the operators in LTL properties (collected by (Pakonen *et al.*, 2016)), where the three most frequently used temporal operators are: *G* (99,5% of all properties), *X* (12,9%) and *F* (5,2%).

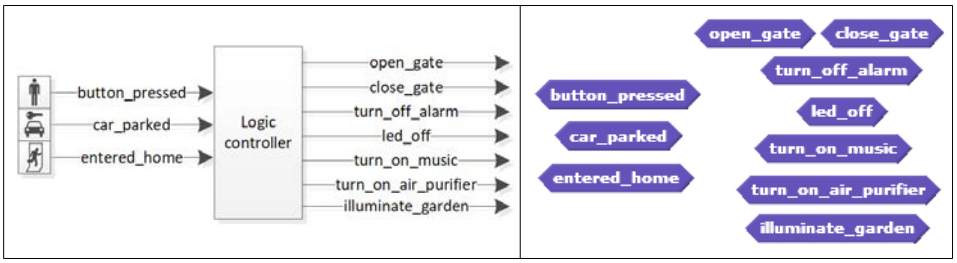
Basic elements of Scratch-based definition of requirements are illustrated in Table 2 (not all elements are by default in the Scratch editor). Using the graphical environment, the only user's task is to drag-and-drop appropriate elements (predefined keywords and specific variables) and to connect them together (put into right places). According to the paper (Aivaloglou and Hermans, 2016), where about 250 thousand Scratch projects were taken into account, 31,5% of the projects use variables, 39,8%, conditional statements and 77,2%, loop statements, so the proposed set of visual elements matches the standard used templates. Different colors (i.e., orange and purple for the main elements and green for logical operators) and shapes (i.e., rounded or angle) of elements simplify the process and prevent any misusage.

Basic graphical temporal logic and Boolean logic elements are transformed into LTL logic elements according to some strictly defined rules as follows in Table 2. Each element has its own equivalent. And so, each Scratch element is transformed directly into appropriate temporal logic operator. Variables remain the same in both interpretations. It should be pointed out, that the negation occurs twice in the visual blocks – the first time as a main element *never*, which refers to situations *never globally* (combined in use as forever-never), and the second time as a simple Boolean operator *not*, which refers to variables (and is used only in combination with variables). Different shapes of the elements simplify the usage of them. It should be also noted, that the Boolean logic operators can be combined together to form, e.g., the triple conjunction.

Let us point out, that whilst the temporal logic elements are constant (regardless of the considered case study), the variables are application specific (defined individually for each case study) and refer to input and output signals of a designed logic controller, taking a Boolean value *true* (signal active) or *false* (signal inactive, combined with the

Table 2
Equivalent elements in Scratch-based definition of requirements and LTL logic

Meaning	Scratch element	LTL logic (and Boolean operators)
forever		*G*
if then		–>
next		*X*
finally		*F*
never (globally)		!
and		&
or		|
not		!



(a) (b)

Fig. 2. Logic controller input and output signals (a) and variables definition in Scratch (b).

Boolean operator *not*). Variables are represented as blue blocks with appropriate signal names. Examples of logic controller input and output signals for a smart home system (a) and the corresponding variables (b) are shown in Fig. 2. The variables are individually defined for the particular case study. Here, input signals are coming from the environment (user action or sensors), while output signals are controlling the real objects. Basic elements together with variables allow definition of the most essential requirements for a designed control system.

3.2. *Complete Expressions*

Let a complete visual representation of a requirement be called *SCRATCH_EXP*. Then, a *SCRATCH_EXP* is formalized as a LTL logic formula, ready to be used in the model checking process (in the proposed approach the nuXmv model checker (Cavada *et al.*,

2014) is used). The complete expressions are created by translating the elements, let an element be called *Ei*. The transformation method of complete requirements in Scratch into temporal logic formulas is described in details in Algorithm 1. It is supposed that the requirement must start with the G or F operator, otherwise a bug is reported. Hence, the never-globally situation is formed with two graphical elements (as it will be shown further in the example).

Algorithm 1. Transformation of complete expressions in Scratch into LTL logic formulas

---

Step1:
write the *LTLSPEC* keyword

Step2:
take the first element $E_1 \in SCRATCH\_EXP$
if E_1 is the *FOREVER* element **then** write the *G* operator
else if E_1 is the *FINALLY* element **then** write the *F* operator
else report an error
end if

Step3:
for all elements $E_i \in SCRATCH\_EXP$, starting from $i = 2$ **do**
if E_i is a basic element **then** write the equivalent operator in LTL logic
else (which means that E_i is a variable) write variable name
end for
write the semicolon (ending the expression)

Step4:
check the complete formula if it is valid (proper structure and sequence of elements)

And so, each Scratch-based defined requirement is translated into LTL logic using the same method (Algorithm 1) and a list of requirements is built to be applied directly in the model checking process. The complete list of requirements is formed according to Algorithm 2. Afterwards, the user's only task is to append the text list at the end of a verifiable file and formal verification can be automatically performed using the nuXmv tool.

Algorithm 2. Transformation of a set of expressions in Scratch into a list of LTL logic formulas

---

for all drawn visual expressions $SCRATCH\_EXP_i$ **do**
apply *Algorithm 1* to each $SCRATCH\_EXP_i$
end for

Let us illustrate the Scratch-based definition of requirements by a sample control process from a smart home system. Smart home systems are used to increase the comfort of inhabitants by applying some automation in daily life. The six sample properties are visually drawn using Scratch, as presented in Fig. 3. The variables are application spe-

cific and have already been shown in Fig. 2a. Let us point out that this is only a snapshot from the sample smart home system and other aspects (e.g., its specification) are here out of scope and are not treated further in this paper.

The requirements in Scratch (Fig. 3) are transformed into a list of requirements according to Algorithm 2 (and involving Algorithm 1). As the result the following list is formed, shown in Fig. 4. The verbal interpretation of requirements can be easily achieved by reading out the visual blocks, assuming that variables related to input and output signals of logic controller are intuitively named, e.g., "Always (forever) if the button is pressed, then next the gate opens" (property 1), "Always (forever) if somebody enters home, then finally the music is turned on and the air purifier is turned on" (property 4) or "Forever never should be possible that both signals for opening and closing the gate are active at the same time" (property 5).

When considering the model checking of any system, the most essential properties concern the safety and liveness of such systems. Safety requirements, stating that the undesired situation will never happen, are presented in LTL temporal logic in the form of *G! property*, which can be written in Scratch as *forever-never* (see Fig. 5). Liveness requirements, stating that the desired situation will eventually happen, are presented in LTL temporal logic in the form of *GF property* (or with an additional condition as *G (condition −> F property)*), which can be written in Scratch as *forever-finally (*or, respectively, *forever-if-then-finally*).

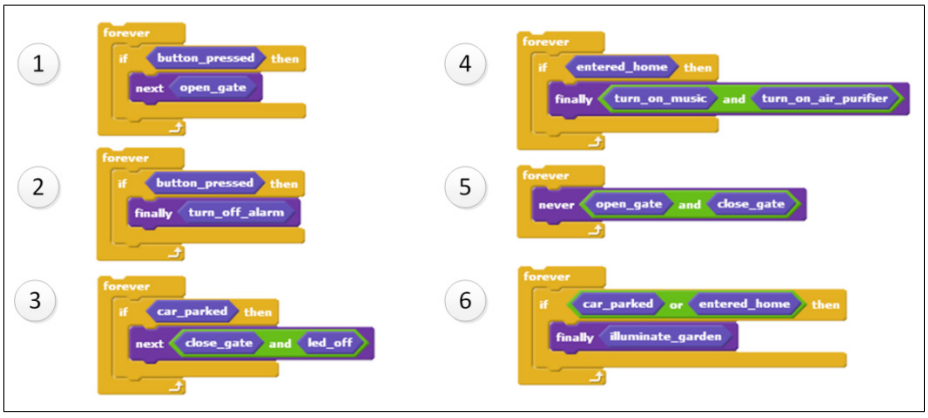


Fig. 3. Scratch-based definition of requirements – examples.

```
LTLSPEC G (button_pressed −> X open_gate);                          -- 1
LTLSPEC G (button_pressed −> F turn_off_alarm);                     -- 2
LTLSPEC G (car_parked −> X (close_gate & led_off));                 -- 3
LTLSPEC G (entered_home −> F (turn_on_music & turn_on_air_purifier)); -- 4
LTLSPEC G !(open_gate & close_gate);                                -- 5
LTLSPEC G ((car_parked | entered_home) −> F illuminate_garden);     -- 6
```

Fig. 4. List of LTL logic requirements from Scratch-based expressions.

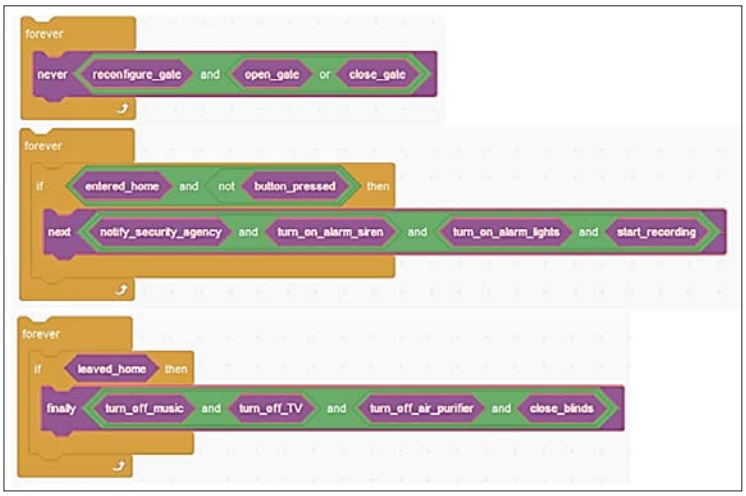


Fig. 5. Scratch-based definition of requirements, focusing especially on safety and liveness.

The sample safety requirement in Scratch is shown in Fig. 5 (at the top) and states: it should never be the case during gate reconfiguration that the control system will be trying to open or close the gate. In turn, the sample liveness requirement in Scratch (Fig. 5, at the bottom) states: whenever the inhabitants leave their home, then finally the comfort maintaining subsystems will be turned off (sometimes not immediately, but after finishing the work cycle, e.g., of an air purifier) and the blinds will be closed. In some situations, it can also be necessary to define that something will happen immediately, as shown in the example (Fig. 5, in the middle), where unauthorized access to the house results in some immediate reactions, such as turning on the alarm siren or notifying a security agency. The Scratch-based requirements from Fig. 5 are transformed into a list of requirements, shown in Fig. 6.

Although the presented examples may seem to be only academic, the real applications of model checking technique in the industry and reported in the literature are also based on such simple patterns. In (Pakonen *et al.*, 2017) the model checking is successfully adopted in the verification of instrumentation and control systems (I&C) in the Finnish nuclear industry. The generalized design issues identified using model checking include, e.g., conflicting operational modes selected on fluctuating input data, with requirements such as that only one mode (*a* or *b*) shall be active at the same time or that

```
LTLSPEC G !(reconfigure_gate & (open_gate | close_gate));
LTLSPEC G ((entered_home & !button_pressed) -> X (notify_security_agency &
          turn_on_alarm_siren & turn_on_alarm_lights & start_recording));
LTLSPEC G (leaved_home -> F (turn_off_music & turn_off_TV & turn_off_air_
          purifier & close_blinds));
```

Fig. 6. LTL logic requirements from Scratch-based expressions, focusing especially on safety and liveness.

the *set_a* command shall reset *b* if signal *c* is not active. The experience in a nuclear industry practice has evidently shown that the main benefit of using the model checking technique is to reveal design issues that are otherwise hard to detect using other methods, such issues as ones related to possible spurious actuation scenarios caused by unintended functionality.

To summarize, in the proposed approach formal verification of logic controllers is more user-friendly than in the standard method of specifying the requirements in temporal logic. In order to perform the model checking process, the Scratch-based requirements, which are simply drawn as visual blocks, are transformed directly into temporal logic formulas. Thus, based on the Scratch code –mathematical formulas are generated. Following the proposed design flow for logic controllers shown in Fig. 1, the considered control process is first specified as a control interpreted Petri net or a diagram of the UML language and then formally written as a rule-based logical model (for more details see Grobelna *et al.*, 2017,). Afterwards, a verifiable code in the nuXmv format (Cavada *et al.*, 2014) is automatically generated and combined with the received requirements in temporal logic. The nuXmv tool then automatically compares the system model with the list of properties and reports whether they are satisfied or not (with generated counterexamples to help localizing the error source). If needed, the control process specification (and so the rule-based logical model) is changed, but sometimes it turns out that also the requirements must be modified. After successful formal verification, the design flow can proceed and the already verified solution can be implemented in a real device.

It should be noted, that the Scratch-based definition of requirements, transformed into temporal logic formulas, is the first of the key elements required for successful formal verification. The second necessary key element here is the system specification (model of the system). With only these two parts (c.f. Fig. 1), the model checking process can be performed. The model of a system, extending in a verifiable file (ready to be checked by a model checker tool) to usually at least hundreds of lines of code, can also be prepared in a more user-friendly way, using, e.g., the rule-based logical model of a control process proposed in (Grobelna *et al.*, 2017). The rule-based logical model (extending to usually only a few dozens of lines) is then automatically transformed into the complete verifiable model of the system (using the implemented *m2vs* tool).

4. Primary Experimental Results

The proposed approach of using Scratch for user-friendly requirements definition of control systems was evaluated by the students of University of Zielona Góra (Poland), Faculty of Computer, Electrical and Control Engineering, in two courses of Discrete Control Systems. The total number of students was 84, most of whom were male students. During classes they learned about various specification techniques of control systems, formal verification including especially the model checking process, requirements definition and temporal logic (specifically it was the LTL and CTL logic). They had also the opportunity to test themselves by interpreting and writing their own requirements for sample real-life control systems.

4.1. *Aim, Method and Results of the Experiment*

The conducted experiment consisted of two parts and its aim was to find out whether Scratch-based definition of temporal requirements for a control system is easier and more user-friendly for students than using the temporal logic itself. The students were given some sample informal specifications of real-life control systems.

Part 1:

Firstly, the students were shown the basics and theory of temporal logic (both the LTL and CTL) in the area of model checking of the embedded specification. Then, the students were asked to describe and explain some sample requirements written in LTL logic. Despite the in-depth earlier presentation of the temporal logic rules, it was difficult for them to correctly decode the existing requirements written as LTL formulas and writing their own ones was even more challenging. In the case of complex formulas, the students had to be assisted by the teacher in order not to make any errors.

Part 2:

Afterwards, the students were introduced to the novel concept of Scratch-based user-friendly definition of requirements for control systems. The presentation of sample properties defined with graphical colorful blocks revived the group and it seemed so attractive, that even students who had not been active so far started to actively participate in the exercises. The Scratch-based method was much easier to use and, what should be emphasized, less error-prone. Moreover, the students had much fun by defining their own requirements for real-life control processes and so the number of specified requirements was much bigger in comparison to the results from the first part of the experiment using only the temporal logic.

Finally, the students were asked which requirements definition method was easier for them to use and more user-friendly. The majority (61%) stated that it was the Scratch-based definition, almost one third (28%) declared that both methods were for them just as easy to use, and the rest (11%) chose temporal logic formulas. The evaluation results are presented on a pie chart in Fig. 7.

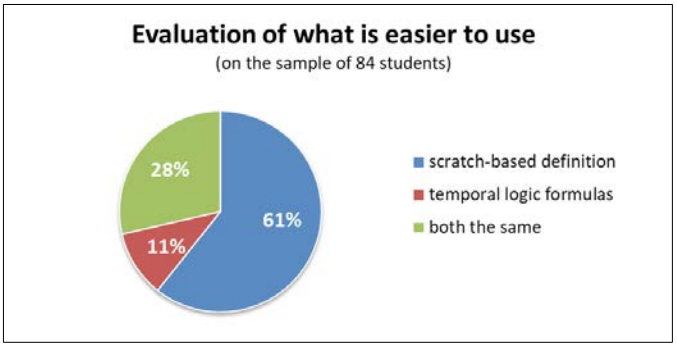


Fig. 7. Experimental results.

4.2. *Conclusions of the Experiment*

The conducted experiment has shown that:

- Students were much more concentrated on requirements definition using Scratch (in comparison to the standard approach using temporal logic).
- Students had no problems in writing and explaining sample requirements (even the complex ones) using the Scratch-based approach.
- Students using Scratch wanted to spend more time on defining further requirements.
- The number of human-related errors in the Scratch-based approach was much lower in comparison to the approach based on temporal logic.

The initial tests confirm that using Scratch is most likely more user-friendly and more effective for non-experienced users in describing the requirements of a control system. Thanks to its graphical representation it is easier to use in comparison to writing directly temporal logic formulas. Also, compared with other requirements specification techniques cited in this paper, using the Scratch-based method requires almost no technical knowledge.

5. Conclusions

The paper introduced a novel user-friendly method of requirements definition for control systems based on the Scratch programming language, in order to simplify the model checking process for the common user and at the same time to achieve a high quality in the designed control system. In brief, the Scratch is easy to use and its simple rules are well understood. The graphical support makes the requirements specification a quick and user-friendly task. In fact, user attention is paid to the requirement itself, and not on the representation of it, which eliminates unnecessary distraction. Using some strictly defined rules, the Scratch-based properties are then translated into temporal logic formulas, making them directly usable in the model checking process.

The initial tests show that the Scratch-based definition of requirements for control systems is simpler and more effective (in terms of the amount of specified properties) in use in comparison to writing directly temporal logic formulas or when using other requirements specification methods for control systems. Additionally, using the visual blocks attracts user attention and results in more requirements being defined. It has been proved to be useful also in teaching students on the topics of temporal logic theory and requirements definition. The proposed method focuses only on the LTL logic, what is its main drawback. However, as explained in the paper, the selected set of elements is sufficient to describe the most used industrial properties found in the literature. Preliminary results presented in this paper confirm the usability of the proposed method. Future research will include further establishment and advancement of the proposed method.

References

Ahmed, T.M., Shang, W. and Hassan, A.E. (2015). An empirical study of the copy and paste behavior during development. In: *IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence*. 99–110. DOI: 10.1109/MSR.2015.17.

Aivaloglou, E. and Hermans, F. (2016). How kids code and how we know: An exploratory study on the scratch repository. In: *Proceedings of the ACM Conference on International Computing Education Research*. 53–61, DOI: 10.1145/2960310.2960325.

Alam, M.R., Reaz, M.B.I. and Ali, M.A.M. (2012). A Review of Smart Homes – Past, Present, and Future. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6), 1190–1203, DOI: 10.1109/TSMCC.2012.2189204.

Asteasuain F. and Braberman, V. (2017). Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2), 239–274, DOI: 10.1007/s00766-015-0242-2.

Autili, M., Inverardi, P. and Pelliccione, P. (2007). Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14(3), 293–340, DOI: 10.1007/s10515-007-0012-6.

Barza, S., Carvalho, G., Iyoda, J., Sampaio, A., Mota, A. and Barros, F. (2016). Model checking requirements. In: *Proceedings of the Brazilian Symposium on Formal Methods*, 10090, 217–234; DOI: 10.1007/978-3-319-49815-7_13.

Beckert B. *et al.* (2017) Generalised test tables: A practical specification language for reactive systems, In: Polikarpova, N. and Schneider, S. (eds) *Integrated Formal Methods, Lecture Notes in Computer Science*. 10510, Springer, Cham, 129–144, DOI: 10.1007/978-3-319-66845-1_9.

Bitterman, N., Shach-Pinsly, D. (2015). Smart home – A challenge for architects and designers, *Architectural Science Review*, 58, 266–274, DOI: 10.1080/00038628.2015.1034649.

Bozzano, M., Cimatti, A., Katoen, J.-P., Katsaros, P., Mokos, K., Nguyen, V., Noll, T., Postma, B. and Roveri, M. (2014). Spacecraft early design validation using formal methods. *Reliability Engineering & System Safety*, 132, 20–35, DOI: 10.1016/j.ress.2014.07.003.

Brill, M. *et al.* (2004). Live sequence charts, integration of software specification techniques for applications in engineering. In: *Lecture Notes in Computer Science, 3147*, 374–399, DOI: 10.1007/978-3-540-27863-4_21.

Campos, J.C. and Machado, J. (2009). Pattern-based analysis of automated production systems. *IFAC Proceedings Volumes*, 42(4), 972–977, DOI: 10.3182/20090603-3-RU-2001.0425.

Cárdenas-Cobo, J., Novoa-Hernández, P., Puris, A. and Benavides, D. (2018). Recommending exercises in Scratch: An integrated approach for enhancing the learning of computer programming, In: Auer, M., Kim, K.S. (eds.) *Engineering Education for a Smart Society, Advances in Intelligent Systems and Computing*, 627, Springer, Cham.

Cavada, R. *et al.* (2014). The nuXmv symbolic model checker, In: Biere, A. and Bloem, R. (eds), *Computer Aided Verification (CAV), Lecture Notes in Computer Science*, 8559. Springer, Cham, DOI: 10.1007/978-3-319-08867-9_22.

Clarke, E.M., Grumberg, O. and Peled, D.A. (1999). *Model Checking*. The MIT Press .

Davis, A. (2005). *Just Enough Requirement Management, Where Software Development Meets Marketing*. Dorset House Publishing Company Incorporated, New York, USA.

Dillon, L.K. *et al.* (1994). A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2), 131–165.

Emerson, E.A. (2008). The beginning of model checking: A personal perspective. In: Grumberg, O. and Veith, H. (eds.), *25 Years of Model Checking: History, Achievements, Perspectives*. Springer Verlag.

Grobelna, I. (2018). Model checking of reconfigurable FPGA modules specified by Petri nets, *Journal of Systems Architecture*, 89, 1–9, DOI: 10.1016/j.sysarc.2018.06.005.

Grobelna, I., Grobelny, M. and Bazydło, G. (2018). User awareness in IoT security. A survey of Polish users. In: *AIP Conference Proceedings*, 2040, 080002, DOI: 10.1063/1.5079136.

Grobelna, I., Wisniewski, R., Grobelny, M. and Wisniewska, M. (2017). Design and Verification of Real-Life Processes With Application of Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(11), 2856–2869, DOI: 10.1109/TSMC.2016.2531673.

Grobelna, I. and Grobelny, M. (2015). UML activity diagrams in requirements specification of logic controllers. In: *International Conference of Computational Methods in Sciences and Engineering (ICCMSE), Greece*. AIP Conf. Proc. 1702, DOI: 10.1063/1.4938882.

Gülbahar, Y. and Kalelioğlu, F. (2014). The effects of teaching programming via Scratch on problem solving skills: A discussion from learners' perspective. *Informatics in Education*, No 1, 33–50.

Hudaib, A., Masadeh, R., Qasem, M. and Alzaqebah, A. (2018). Requirements prioritization techniques comparison. *Modern Applied Science*, 12(2), DOI: 10.5539/mas.v12n2p62.

Huth, M. and Ryan, M. (2004) Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press.

Jiang, Y., Liu, J., Dowek, G. and Ji, K. (2018) Towards Combining Model Checking and Proof Checking, The Computer Journal, DOI: 10.1093/comjnl/bxy112.

Klimek, R. (2018) Visualization of logical formulas, Proceedings of the Federated Conference on Computer Science and Information Systems, 15, 419–424, DOI: 10.15439/2018F264.

Klimek R., and Rogus, G. (2015) Proposal of a Context-Aware Smart Home Ecosystem, In Rutkowski L. *et al.* (eds), Artificial Intelligence and Soft Computing, ICAISC, Lecture Notes in Computer Science, 9120, Springer, Cham.

Kropf, T. (1999) Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems. Springer.

Lengyel, L. *et al.* (2015) Quality Assured Model-Driven Requirements Engineering and Software Development, The Computer Journal, 58, Issue 11, 3171–3186, DOI: 10.1093/comjnl/bxv051.

Moreno, J. and Robles, G. (2014) Automatic detection of bad programming habits in scratch: A preliminary study, Frontiers in Education Conference (FIE), IEEE, DOI: 10.1109/FIE.2014.7044055.

Naumchev, A. and Meyer, B. (2017) Seamless requirements, Computer Languages, Systems & Structures, 49, 119–132, DOI: 10.1016/j.cl.2017.04.001.

Nergaard, H., Ulltveit-Moe, N. and Gjosæter, T. (2015) A scratch-based graphical policy editor for XACML, International Conference on Information Systems Security and Privacy, IEEE, 1–9.

OMG Unified Modeling Language, resources available at www.uml.org (last accessed 2019-09-26).

Ouahbi, I. *et al.* (2015) Learning basic programming concepts by creating games with scratch programming environment, Procedia – Social and Behavioral Sciences, 191, 1479–1482, DOI: 10.1016/j.sbspro.2015.04.224

Ozoran, D., Cagiltay, N. and Topalli, D. (2012) Using scratch in introduction to programming course for engineering students, 2nd International Engineering Education Conference (IEEC).

Ölveczky, P.C. (2017) Formalizing and Checking Requirements. Designing Reliable Distributed Systems. Undergraduate Topics in Computer Science, Springer, London, DOI: 10.1007/978-1-4471-6687-0_16.

Pakonen, A., Tahvonen, T., Hartikainen, M., Pihlanko, M. (2017) Practical applications of model checking in the Finnish nuclear industry, Proceedings of the 10th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies (NPIC & HMIT 2017), San Francisco, CA, USA, pp. 1342–1352, American Nuclear Society.

Pakonen, A., Pang, C., Buzhinsky, I. and Vyatkin, V. (2016) User-friendly formal specification languages – conclusions drawn from industrial experience on model checking, Proceedings of IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), DOI: 10.1109/ETFA.2016.7733717.

Resnick, M. *et al.* (2009) Scratch: programming for all, Communications of the ACM, 52, Issue 11, 60–67, DOI: 10.1145/1592761.1592779.

Robertson, S. and Robertson, J. (2012) Mastering the Requirements Process: Getting Requirements Right, Addison-Wesley.

Sáez-López, J.-M., Román-González, M. and Vázquez-Cano, E. (2016) Visual programming languages integrated across the curriculum in elementary school: A two year case study using "Scratch" in five schools, Computers & Education, 97, 129–141.

Scratch Project Editor, available online at scratch.mit.edu (last accessed 2019-09-26).

Smith, M.H., Holzmann, G.J. and Etessami, K. (2001) Events and constraints: a graphical editor for capturing logic properties of programs, Proceedings of the 5th IEEE International Symposium on Requirements Engineering, DOI: 10.1109/ISRE.2001.948539.

Tempelmeier, T. (2011) Proving the Safety of Autonomous Systems with Formal Methods – What Can You Expect?, In Unger, H., Kyamakya, K. and Kacprzyk, J. (Eds.) Autonomous Systems: Developments and Trends, Studies in Computational Intelligence, 391, Springer-Verlag.

Tudor, D. and Walter, G. A. (2006) Using an agile approach in a large, traditional organization, Proceedings of AGILE 2006, Minneapolis, IEEE, DOI: 10.1109/AGILE.2006.60.

Vyatkin, V. and Bouzon, G. (2008) Timing Diagrams as Visual Specifications in Verification of Industrial Automation Controllers, Journal on Embedded Systems, DOI: 10.1155/2008/251957.

Wiśniewski, R. (2018) Dynamic partial reconfiguration of concurrent control systems specified by Petri nets and implemented in Xilinx FPGA devices, IEEE Access, 6, 32376–32391, DOI: 10.1109/ACCESS.2018.2836858.
Wisniewski, R. and Grobelna, I. (2018) Design of Multi-Context Reconfigurable Logic Controllers Implemented in FPGA Devices Oriented for Further Partial Reconfiguration, Journal of Circuits, Systems and Computers, 27, Issue 6, DOI: 10.1142/S021812661850086X.
Woodcock, J., Larsen, P. G., Bicarrequi J. and Fitzgerald, J. (2009) Formal methods: Practice and experience, Journal ACM Computing Surveys, 41, Issue 4, Article no.19.
Ying, M. and Feng, Y. (2018) Model-checking quantum systems, National Science Review, DOI: 10.1093/nsr/nwy106.

I. Grobelna is an Assistant Professor in the Faculty of Computer, Electrical and Control Engineering at the University of Zielona Góra, Poland. She received her PhD in Computer Science from the same university in 2012. Her research interests include design, specification and verification of embedded digital systems, especially considering the human aspect. She is an author of 3 books and over 50 technical papers in peer-reviewed journals and conferences.