An intelligent tutor to learn the evaluation of microcontroller I/O programming expressions

Hugo Arends
HAN University of Applied Sciences and
Open University of the Netherlands
hugo.arends@han.nl

Bastiaan Heeren
Open University of the Netherlands
bastiaan.heeren@ou.nl

Hieke Keuning
Windesheim University of Applied Sciences and
Open University of the Netherlands
hw.keuning@windesheim.nl

Johan Jeuring
Utrecht University and
Open University of the Netherlands
j.t.jeuring@uu.nl

ABSTRACT

Embedded systems engineers need to learn how I/O programming expressions for microcontrollers evaluate. We designed, implemented, and tested an intelligent tutoring system prototype for learning such evaluations. The Microcontroller Knowledge (MicK) tutor guides a student step-by-step towards a solution. A domain reasoner, built using the Ideas framework, generates feedback and hint messages. MicK supports various microcontrollers and programming languages by dynamically creating exercises and using lookup environments. Instructors can easily customise MicK, for instance by adding new exercises and changing the reported feedback messages. MicK is validated in a pilot study with questionnaires filled in by students and lecturers. The results show that the step-by-step feedback and hint messages contribute to understanding how microcontroller I/O programming expressions evaluate.

CCS CONCEPTS

• **Social and professional topics** → **Computer engineering education**; • **Applied computing** → *Interactive learning environments*;

KEYWORDS

intelligent tutoring system, domain reasoner, automated feedback, programming tutor, expression evaluation, microcontroller

ACM Reference Format:
Hugo Arends, Bastiaan Heeren, Hieke Keuning, and Johan Jeuring. 2017. An intelligent tutor to learn the evaluation of microcontroller I/O programming expressions. In *Proceedings of Koli Calling 2017*. ACM, New York, NY, USA, 8 pages. https://doi.org/https://doi.org/10.1145/3141880.3141884

Koli Calling 2017, November 16–19, 2017, Koli, Finland

ACM ISBN ACM ISBN 978-1-4503-5301-4/17/11…$15.00
https://doi.org/https://doi.org/10.1145/3141880.3141884

1 INTRODUCTION

Today's connected society uses the internet of things, robotics, virtual home assistants, smart systems, and more. These systems are implemented using microcontrollers to get information from sensors, to manipulate this information, and to control actuators based on this information. To build microcontroller-based systems, students need both hardware and software skills. Such skills are taught in embedded systems courses both at the bachelor and master level, and in online communities.

Learning is most effective when an expert teacher provides one-to-one instruction [2]. With an increasing popularity of embedded systems courses, both on campus and online, providing such individual instruction is hard, if not impossible to realise. Intelligent tutoring systems (ITSs) might offer a solution to this problem. An ITS is a software application designed to simulate a human tutor's behaviour, and it can support a student's learning process in many ways [16].

One of the many challenges in software development for microcontrollers is related to I/O programming, such as manipulating bits in hardware registers and using loops to wait for the logical change of a single bit. The evaluation of these expressions into a normal form can be hard to comprehend for novice embedded systems engineers. An example of such an evaluation is presented for the following expression:

```
PORTB = 2 & (0b00000001 << x);
```

In this expression, `PORTB` is a hardware register for a specific microcontroller, which is assigned a value that depends on the value of variable `x`. Assume that variable `x` is equal to 3. This expression can be written into normal form with the evaluation steps presented below in *italics*. The intermediate result after an evaluation step is presented for clarity.

Substitute variable `x` *with 3*

```
PORTB = 2 & (0b00000001 << 3);
```

Evaluate the bitwise left shift operator `(<<)`

```
PORTB = 2 & 0b00001000;
```

Evaluate the bitwise AND operator `(&)`

```
PORTB = 0;
```

This example shows that if `x` is equal to 3, `PORTB` will be assigned the value 0. It is important to understand such top-down evaluations

of complex expressions for learning programming [10], especially for learning I/O programming in embedded systems.

This paper describes the design of an ITS for learning the evaluation of microcontroller I/O programming expressions. The ITS guides students step-by-step towards a solution by providing feedback and hint messages. A characteristic of the domain is the diversity in microcontrollers and programming languages. The main contribution of the research project is the creation of a single ITS that handles this diversity, rather than a separate ITS for each variant. The second contribution is a small-scale pilot study with participants with different domain knowledge. The goal of the pilot study is to determine if the step-by-step guidance offered by the ITS contributes to understanding how microcontroller I/O programming expressions evaluate.

Section 2 describes related work. Section 3 provides a system overview and demonstrates how students and instructors interact with the system. Section 4 describes design decisions for the implementation. Section 5 discusses how we validated the ITS. Section 6 concludes.

2 RELATED WORK

This section discusses work related to teaching the evaluation of programming expressions within the context of microcontroller I/O and in a broader programming context.

2.1 Teaching expression evaluation

Bitwise and logic operators, such as bitwise AND (&) and negation (!), are key concepts in microcontroller I/O programming that are explicitly taught in embedded systems courses. For example, the book about microcontroller basics by Davies [4] discusses the important aspects of C for embedded systems, such as declarations, shifts, low-level logic operations, masks to test and modify individual bits, bitfields, and unions. They are explained by means of simple examples. Dolman [5] visualises the step-by-step evaluation of complex composite expressions. Pardue [12] uses a similar visualization technique to explain the evaluation of single bitwise and logic operators. In a discussion on the evaluation of composite expressions he notes that if his explanation is not clear, the reader should use a pencil and paper and work through it until it is, emphasising the importance of understanding the step-by-step evaluation of such statements.

2.2 Tools for learning expression evaluation

The stepper functionality of a debugger allows inspecting the evaluation of a program. At every step, a student sees the effect of executing a statement on the internal state of a microcontroller. In case of a composite expression, only the resulting value is shown, and not the step-by-step evaluation. Debuggers are not designed for student learning and do not give feedback or hints on the evaluation of programs and expressions.

However, several visualisation tools exist to support students in understanding the run-time behaviour of computer programs. Sorva et al. [14] provide an extensive review of *generic* visualisation systems that teach how the execution of programs works. The authors consider visualisation of expression evaluation as a specialisation and exclude those tools from their review, although some generic systems can also show expression-level visualisations. Most of these systems offer controlled viewing, meaning that the student is in control of how he or she views the visualisation, but does not have any other active task. Sirkiä [13] describes the need for expression-level visualisations in addition to line-level visualisation, because it provides students with more insight into how a program works.

An example of a visualisation tool that offers more interactivity is UUhistle [15], a tool for visualising Python programs. Students can play the role of the computer by dragging and dropping elements to create variables, assign and retrieve values, and by clicking buttons to invoke operators and methods. The tool can detect errors and allows the student to ask for a hint. A difference between our tutor and UUhistle is the different domains that they support, making our tutor a more specialised system. Our system focusses more on how operators work by letting students fill in the results of invoking an operator instead of doing it for them, and giving hints and feedback on their actions. This is especially important because domain-specific tasks such as applying bitwise operators and doing number conversion are difficult for students.

WADEIn II [3] is an adaptive and explanatory visualisation tool for C programming expressions, supporting several language constructs and operators. The tool lets students explore the evaluation of expressions by watching animations and reading textual descriptions. Students are also given the option to actively solve problems by indicating the evaluation order and providing values of subexpressions. The tool behaves adaptively by decreasing animation speed and hiding parts of the description as the student progresses. The tool does not offer any hints to students.

Kumar [10] describes a tutoring system for learning how to evaluate C++/Java programming expressions step-by-step. The tutor helps students to learn the evaluation of expressions by generating a problem, letting the student solve the problem, and assessing the solution. New expressions can be generated randomly, or added by an instructor. The tutor ensures that a student never sees the same problem twice. The tutor generates feedback using colours that indicate whether or not a student selects the correct subexpression and correctly calculates the subexpression's evaluation step. If a student does not know how to proceed, the student can stop the task and the tutor presents the worked-out solution with detailed feedback for each evaluation step. The main differences between Kumar's tutor and our tutor are that our tutor provides detailed hints and solutions during each evaluation step, our tutor allows students to take multiple evaluation steps at once, and in our tutor students can submit their own expressions.

Olmer et al. [11] developed a prototype of a tutor for evaluating Haskell expressions. The prototype supports a student in the understanding of programming concepts and evaluation strategies by showing the step-by-step evaluation of expressions. It provides feedback after each evaluation step. If a student does not know how to proceed, hints are provided by the tutor, such as the number of steps left, or all rules that can be applied according to the evaluation strategy. The tutor is intended for the functional programming paradigm, whereas our tutor is intended for the imperative programming paradigm and specifically for microcontroller I/O programming.

3 AN EXAMPLE SESSION

This section demonstrates our ITS prototype MicK. Section 3.1 introduces the web application front-end and describes typical student interactions. Section 3.2 presents the customisation options for instructors.

3.1 Student interaction

Students interact with MicK through a web application[1], as shown in Figure 1. An evaluation task is started by selecting a microcontroller and programming language from a dropdown box. Upon selection, relevant example expressions are automatically added to another dropdown box. A student then either selects an initial expression from the examples and optionally changes it, or manually enters an initial expression. Assume that a hypothetical student wants to evaluate the following expression for the ATmega328P microcontroller and the ANSI-C programming language. The purpose of this expression is to poll bit `UDRE0` in register `UCSR0A` until it is logic one:

```
while( ! (UCSR0A & (1 << UDRE0)) ) { ; }
```

The end of a typical evaluation task for this expression is shown in Figure 1. The remainder of this section discusses how the student arrives at this result using the step-by-step guidance provided by MicK.

As soon as the student starts the task, MicK first analyses the initial expression and presents a value for register `UCSR0A` and for definition `UDRE0`:

> *Values of definitions, registers and volatile variables for this microcontroller and programming language:*
>
> *UCSR0A = 0b00001111*
> *UDRE0 = 5*

All information required to solve the evaluation task is now available for the student. A second input field is added for the student to enter the next evaluation step. The student might choose to substitute both the register and definition at once:

```
while( ! (0b00001111 & (1 << 5)) ) { ; }
```

Clicking the Validate button validates this evaluation step. MicK responds:

> *That is correct.*

Another input field is added to the list of evaluation steps, which allows the student to enter the next evaluation step. The student enters the next step by evaluating the shift left operator, but makes a common mistake by reversing the order of operands of the shift left operator:

```
while( ! (0b00001111 & 10) ) { ; }
```

The student clicks the Validate button and MicK responds with the message:

> *That is incorrect. The operands of the shift left operator are reversed.*

This message tells the student exactly what the mistake is. By clicking the hyperlink 'shift left', a new browser window is opened with an external website that explains the shift left operator. Suppose the student still does not know how to proceed and clicks the Hint button. MicK clears the incorrect input field and responds with the message:

> *1*
> *⇒ rewrite this decimal number to its binary representation*

This message tells the student that, prior to evaluating the shift left operator, it is more convenient to first rewrite the decimal value 1 to its binary representation . Again, the hyperlink points to an external website for more information about the operation. Let us assume that the student still does not know how to proceed and clicks the Show button. MicK responds with the message:

> *1*
> *⇒ rewrite this decimal number to its binary representation*
> *0b00000001*

MicK also fills in the solution to this step in the input field:

```
while( ! (0b00001111 & (0b00000001 << 5)) ) { ; }
```

The student clicks Validate and MicK responds with the message:

> *That is correct.*

Now the student makes a mistake by entering and validating the next expression:

```
while( ! (0b00001111 & 0) ) { ; }
```

MicK responds with the message:

> *Although the expressions are equivalent, you took a wrong step.*

This message tells the student that the submitted expression evaluates to the same normal form, but that the evaluation step is incorrect. No new input field will be added to the evaluation list. The student might edit the expression, or request a hint, or request the solution to this step. Let us assume the student edits the expression:

```
while( ! (0b00001111 & 0b00100000) ) { ; }
```

The student clicks the Validate button and MicK responds with the message:

> *That is correct.*

The student now tries to take multiple steps, but forgets a closing parenthesis:

```
while( ! (false ) { ; }
```

The student clicks the Validate button and MicK responds with the message:

> *Syntax error 1:1:*
> *unexpected 'w'*
> *expecting '(', ')', '{', end of input, or operator*

The student corrects the mistake:

```
while( ! (false) ) { ; }
```

The student clicks the Validate button and MicK responds with the message:

> *That is correct.*

The student enters the final step:

```
while( true ) { ; }
```

The student clicks the Validate button and MicK responds with the message:

> *That is correct.*
> *You have finished the task successfully!*

[1] http://ideas.cs.uu.nl/mick

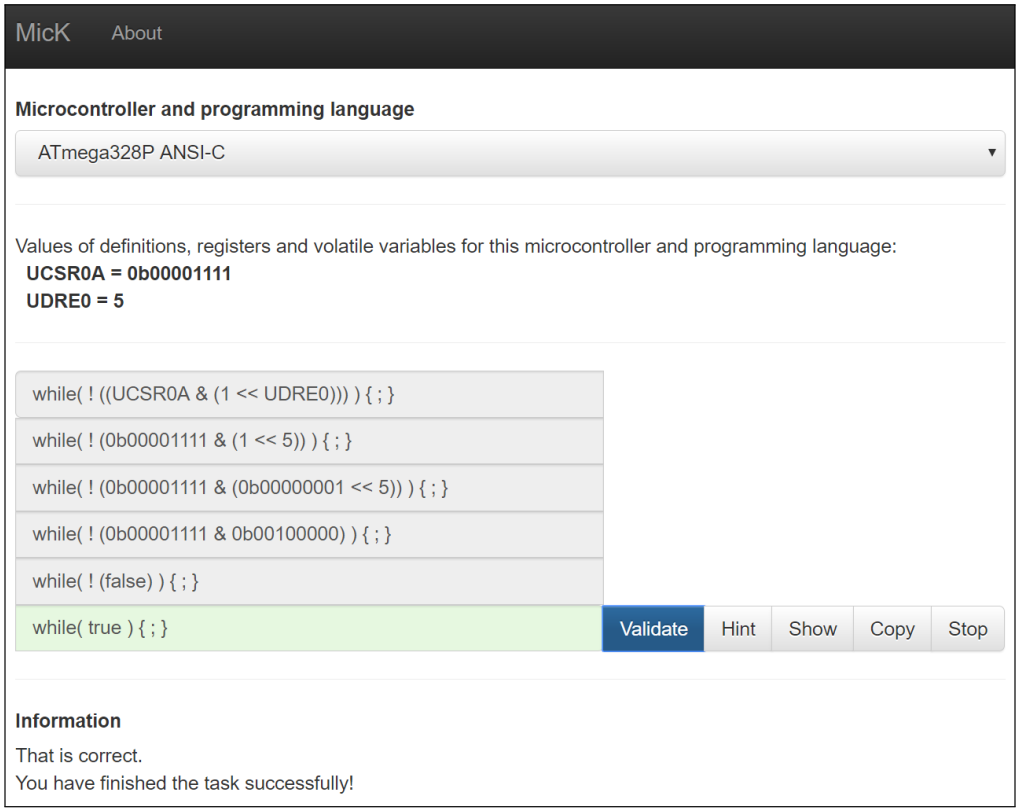

Figure 1: The student has finished the task after five successful evaluation steps.

This message tells the student that the task is finished and that the expression is in a normal form. The input field with the final solution is marked green and no new input field is added to the evaluation list.

3.2 Instructor interaction

Figure 2 presents a system overview of MicK. The web application communicates with a domain reasoner through the internet. A domain reasoner [6] is a software application that is capable of reasoning about a problem domain, and runs on a web server. The details of this domain reasoner are described in Section 4.

Figure 2 shows that instructors can customise MicK by adding or modifying several files. An instructor places these files in a folder on the web server's filesystem and the files are automatically read by the domain reasoner. Each exercise configuration file is used to configure a new exercise. MicK automatically adds these exercises to the list of exercises the student can choose from in the web application. For changes to take effect, a client only needs to reload the web application.

An exercise configuration file describes the following options:

- Exercise id: unique exercise identification.
- Examples: zero or more example expressions that appear in the examples dropdown box in the web application.
- Initial values: initial values for registers and volatile variables, such as `UCSR0A` in the example in Section 3.1. An instructor provides these values to steer the students' learning process.
- Word length: the number of bits of the architecture for the selected microcontroller used for zero padding numbers in binary and hexadecimal number representation.

Furthermore, a configuration file stores the paths to three more files, which are shown as dashed lines in Figure 2:

- Feedback script: the textual feedback and hint messages provided by MicK are specified in a script file. Thus an instructor can easily change feedback and hint messages, for instance to match classroom lectures, to use hyperlinks for specific websites, or to provide feedback in a specific language.
- Programming language definitions: an instructor can change the programming language by specifying language-specific keywords and tokens.
- Microcontroller definitions: microcontroller-specific definitions, such as `UDRE0` in the example in Section 3.1, which are often provided by a microcontroller vendor, optionally in different file formats.

4 AN ITS FOR MICROCONTROLLER I/O PROGRAMMING

MicK consists of two components: a web application and a domain reasoner. The web application is the user interface for students and

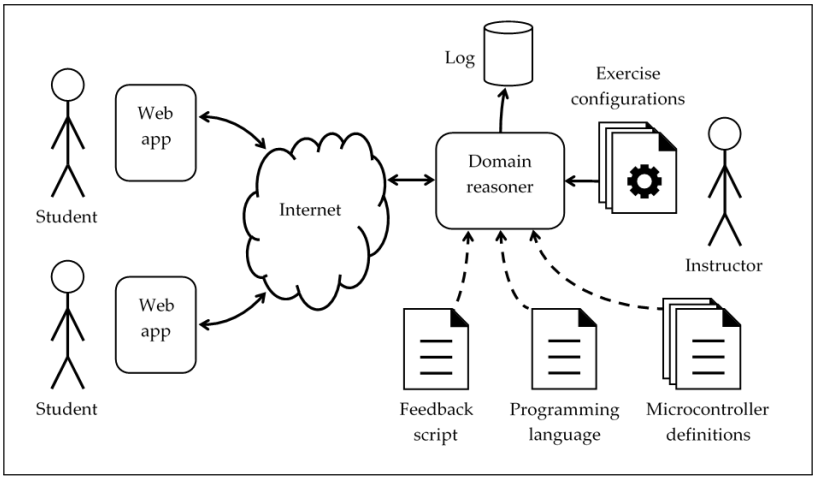

Figure 2: System overview of MicK, showing the building blocks, the information flow, and configuration options.

is implemented using Bootstrap, JQuery and JavaScript. The web application uses feedback services offered by the domain reasoner. The remainder of this section discusses the design decisions for the domain reasoner, which is based on the Ideas framework [7].

4.1 Exercises

Feedback services [7] are generic services that can be used for any problem domain. An example of such a service is the 'onefirst' service, which calculates a possible next step. The feedback services use domain-specific components for calculating feedback messages. The most important components are the domain data type, the rules, and the strategies.

Data type. A data type is used for an abstract representation of domain-specific expressions. The data type for this domain is based on the grammar of typical microcontroller I/O programming expressions. The following grammar is implemented in MicK:

```
expr ::= stmt
       | stmt `;'
       | stmt `;' expr
       | `{' expr `}'
stmt ::= `skip'
       | identifier `assign' op1
       | `while' `(' op1 `)' `{' expr `}'
       | op1
op1  ::= op1 `|' op2  | op2
op2  ::= op2 `&' op3  | op3
op3  ::= op3 `<<' op4 | op4
op4  ::= op4 `+' op5  | op5
op5  ::= `!' op6      | op6
op6  ::= `(' op1 `)'  | num | bool | identifier
num  ::= dec | bin | hex
bool ::= `true' | `false'
```

A parser is implemented for parsing sentences belonging to this grammar to the value of the abstract data. A pretty printer is implemented for turning the abstract data into a human readable string.

Rules. A rule defines how a value of the data type can be transformed, and rules can therefore be used to evaluate simple expressions. For example, a rule for the infix operator '&' calculates the bitwise AND of both operands. We define five groups of rules for the domain of microcontroller I/O programming:

- Infix operators: defines an operation that takes two operands.
- Prefix operators: defines an operation that takes a single operand.
- Number representation: defines how numbers are converted to their decimal, binary, hexadecimal, and Boolean representations.
- Substitution: defines how to substitute a register or volatile variable by an exercise-specific value.
- Buggy rule: defines a common mistake for any of the above transformations.

Strategies. A strategy combines rules to solve multi-step exercises. A strategy is expressed in an embedded domain-specific language (EDSL) that is interpreted by the Ideas framework as a context-free grammar. A domain reasoner uses strategies to calculate feedback messages [8].

A strategy can be composed out of rules, but also out of other strategies. Strategy combinators combine strategies. For example, the sequence combinator (. ∗ .) puts two strategies in sequence. As an example, the strategy for the bitwise AND operator is defined as follows:

```
bitwiseAndS :: ExprStrategy
bitwiseAndS =
  check isBitwiseAnd
  .*. convOperands toBinS toBinS
  .*. bitwiseAndRule
```

The strategy first checks if the bitwise AND operation is applicable. The strategy then makes sure that the left and right operand are represented in the binary representation, by using the toBinS substrategy. Finally, if these preconditions are met, the strategy applies the bitwise AND rule.

The domain reasoner for MicK implements two top-level strategies: one for rewriting expressions containing an assignment, and one for rewriting expressions containing a while-statement. Both strategies follow a bottom-up procedure in which they first try to either substitute a variable, or calculate the result of an operator. After one of these has been applied, the strategy stops and starts over again, bottom-up. This continues until none of the two strategies can be applied anymore, resulting in a normal form for expressions with an assignment. The strategy for rewriting expressions with a while-statement into normal form, continues by rewriting the condition of the while-statement to a Boolean representation and, if applicable, evaluates any logical operation.

4.2 Customisable exercises

MicK supports multiple microcontrollers and programming languages by dynamically generating exercises from configuration files at start-up of the domain reasoner, as illustrated in Figure 2. By dynamically generating exercises, there is no need for programming or recompiling the domain reasoner.

Each exercise is specific for a particular microcontroller. The microcontroller-specific definitions are provided in files, such as C-header files. MicK represents these definitions and their values as a list, which is used by the domain reasoner as a lookup environment whenever a definition must be substituted.

MicK realises support for multiple programming languages by allowing instructors to customise keywords and tokens from the grammar in a language definition file. Typical expressions for microcontroller I/O programming do not require different abstract syntax structures. Similar to the microcontroller definitions, the tokens and their values are stored in a lookup environment. MicK uses this lookup environment for parsing and for pretty printing.

4.3 Feedback generation

Whenever a student asks for a hint or validates an expression, MicK calculates a textual feedback message. An instructor can customise this message in a script file. We use the eleven categories for feedback generation for learning programming described by Keuning et al. [9] to determine what feedback and hint messages must be supported by MicK. From these eleven subcategories the following five subcategories, which contribute best in helping students to understanding the evaluation of microcontroller I/O programming expressions, are supported:

- Task-processing rules (TPR). To help a student getting started, it must be clear how to approach a task. This is realised by providing a feedback message when a new task is selected in the web application.
- Explanations on subject matter (EXP). If a student makes a mistake, or does not know how to proceed towards a solution, information is provided on the subject matter. The feedback messages in Section 3.1 showed this by providing a hyperlink to an external website with information about the subject.
- Compiler errors (CE). An expression is checked for syntactic errors and non-existing definitions. Section 3.1 showed an example of a detailed error message which is provided by the parser to help students solve syntactic mistakes.
- Solution errors (SE). If the evaluation step submitted by a student is not correct, the feedback service calculates a feedback message indicating that the student made a mistake. An example of such a feedback message in Section 3.1 is: *Although the expressions are equivalent, you took a wrong step.*
- Task-processing steps (TPS). To proceed towards a solution, MicK provides detailed feedback about a next possible evaluation step.

We use the domain reasoner's diagnose service for generating feedback messages when the student validates an expression. The default diagnose service calculates, amongst others, semantic equivalence of two expressions to determine correctness of the evaluation step. This is a problem for the domain of microcontroller I/O programming, because incorrect evaluation steps might still lead to semantically equivalent expressions. The following two expressions, for example, are semantically equivalent, because they both evaluate to `PORTB = 0;`. The evaluation step, however, is incorrect:

```
PORTB = 2 & (0b00000001 << 3);
```

Incorrect evaluation of operator `<<`

```
PORTB = 2 & (0b00000000);
```

We have created a custom diagnose service to solve this problem. An additional relation called 'semantically equivalent delta pairs' is calculated when two expressions are submitted for diagnosis. A delta pair is a maximum subexpression that is different when two expressions are syntactically compared. The delta pair for the two expressions in the example is `(0b00000001 << 3, 0b00000000)`. A detailed description of this relation and the custom diagnose service is given in the first author's master's thesis [1].

5 VALIDATION

To validate our prototype tutor, we have organised experiments with 46 participants in total. We asked two groups of students with different prior knowledge to participate in an experiment. The first group of 25 fourth year Electrical and Electronic Engineering bachelor students from HAN University of Applied Sciences in the Netherlands participated on December 20th, 2016. The second group of 18 first year bachelor students participated on January 9th and 12th, 2017. Besides these students, three lecturers from the same university participated in the experiment on January 16th, 2017. Prior to each experiment, the participants received a 15 to 20 minute classroom instruction on expression evaluation, tutoring systems in general, and how to use MicK. After the introduction, the participants were pointed to online questions. They answered these questions independently in half an hour. The experiment consisted of the following parts:

- Answering introductory questions, e.g. what year a student is in.
- Using MicK to rewrite microcontroller I/O programming expressions into normal form. Participants could select three exercises, and for each exercise three different example expressions. Participants were encouraged to modify the examples or to use their own expression.
- Answering concluding questions, e.g. how useful is the possibility to ask for a hint?

Table 1: Average results from closed questions on a five-point Likert scale.

Question	*Average*
When MicK is first started it is clear how to start with a task.	4.0
How useful is a feedback message to you after each step?	3.7
How useful is it to make the tutor show the solution to a step?	4.3
How useful is it to be able to ask for a hint?	4.2
In case you have asked for one or more hints, to what extent did they help you to solve the task?	3.8

5.1 Results

We conducted the experiments to get an initial idea of how users appreciate the generated feedback messages, and if the feedback messages contribute to understanding how microcontroller I/O programming expressions evaluate. We obtained data by asking closed questions on a five-point Likert scale, by asking open questions, and by analysing log files.

Closed questions. Table 1 presents the average results of the answers to the closed questions.

A score of 1 corresponds to the Likert level 'not', and a score of 5 to 'very'. The results show that:

- participants find the possibility to make MicK show the solution more useful than the ability to ask for a hint;
- feedback messages with information about the next step, such as hints, help students towards a solution;
- feedback messages after each step and the possibility to ask for the solution to a step help students towards a solution, and therefore help them to understand the evaluation of microcontroller I/O programming expressions.

Open questions. We asked participants to recommend improvements. Some suggest that if a step is diagnosed as not obvious or incorrect, MicK should explain why. This suggests to also implement the feedback subcategory error correction (EC) [9]. Furthermore, feedback messages related to compiler errors seem hard to understand. The messages are probably not descriptive enough to help students with solving syntactic mistakes.

We also asked participants what they like about MicK. Many answers mention the step-by-step explanations and how this helps in understanding the problem domain.

Log analysis. The domain reasoner logs all interactions in a database. Table 2 presents the results of these interactions broken down by group.

The participants started 499 new tasks of which 221 expressions contained a syntax error. A total of 1532 expressions were submitted for validation, of which 116 expressions contained a syntax error and 1416 expressions were diagnosed. A detailed description of each diagnosis is available in the first author's master's thesis [1]. The results show a remarkably high percentage of syntax errors, especially for expressions that are submitted when a new task is started. The majority of these syntax errors is made by first year students, which is presumably related to lesser experience. For probably that same reason, the diagnosis 'small rewrite step, not recognised' shows that first year students submit relatively more similar expressions for validation. The participants asked more often for the solution to a step (704 times), than for a hint (403 times).

5.2 Threats to validity

All students participating in the experiment were taking a course from the first author at the moment the experiments were performed. Participation was on a voluntary basis. Another threat to generalising the results is that only three instructors participated, all of them colleagues of the first author. However, by selecting participants with different domain and prior knowledge the results are not limited to a single group.

The results show that the step-by-step guidance towards a solution is considered useful by students. We did not expect that students also want to have feedback messages in the error correction subcategory, because students already have the possibility to ask for a hint. A threat to the validity of these conclusions is that they are based on interpretation of the student answers.

6 CONCLUSION

This paper describes the usage, design, implementation, and validation of an ITS prototype for the domain of microcontroller I/O programming. Key concepts in programming for this domain are bitwise and logic operators, hardware registers, and microcontroller specific definitions. The overall conclusion is that we have created a single ITS that supports a diversity of microcontrollers and programming languages and that the step-by-step guidance helps students understand how microcontroller I/O programming expressions evaluate.

We have used the Ideas framework to implement a domain reasoner. We have defined five groups of rules for data type transformations, and specified strategies for rewriting typical microcontroller I/O programming expressions into normal form.

The ITS dynamically generates exercises from configuration files. Microcontroller-specific definition files are parsed and the definitions are stored in a lookup environment, which is used whenever a substitution is required. The ITS supports multiple programming languages by allowing instructors to customise tokens from the grammar. These tokens are stored in a lookup environment, which is used during parsing and pretty printing.

From the answers of the students to the questions asked in the experiment we conclude that it helps students to understand the evaluation of microcontroller I/O programming expressions when the feedback messages explain the subject matter (EXP), solution errors (SE), and task-processing steps (TPS). Students want to have

Table 2: Results from log database analysis broken down by group.

Log characteristic	***Total***		*1st year*	*4th year*	*Instr.*
New task started	**499**		292	198	9
Syntax error in initial expression	**221**		172	46	3
Valid initial expressions	**278**		120	152	6
Expressions validated	**1532**		566	928	38
Syntax error in submitted expression	**116**		45	67	4
Diagnosis	**1416**	(100%)	521	861	34
Not equivalent, unknown mistake	**190**	(13%)	64	124	2
Common mistake with buggy rule	**2**	(0%)	0	1	1
Small rewrite step, not recognised	**52**	(4%)	32	20	0
Rewrite step expected by expert strategy	**840**	(59%)	347	479	14
Correct step, but detour from strategy	**18**	(1%)	6	11	1
Equivalent, wrong step, unknown mistake	**145**	(10%)	33	106	6
Equivalent, correct step, but unknown	**169**	(12%)	39	120	10
Hint for current step requested	**403**		158	239	6
Show solution to current step requested	**704**		313	387	4

more detailed feedback messages related to error correction (EC), such as syntactic mistakes.

The results of the pilot study look promising, even though MicK implements a very small set of programming expressions for the specific domain of microcontroller I/O programming. With support for more programming expressions, further research should explore the tutor's effect on student learning. Furthermore, MicK uses a web application for students to interact with, not a development environment or actual microcontroller. It would be interesting to explore the idea of combining an expression evaluator with the stepper functionality of a debugger. This allows students to learn the evaluation of expressions in the context of an executing program. Adding a student model for tracking a student's progress and selecting tasks is also an important area for future work.

REFERENCES

[1] H. Arends. Intelligent tutor to learn the evaluation of microcontroller I/O programming expressions. Master's thesis, Open University of the Netherlands, 2017.

[2] B.S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher*, 13(6):4–16, 1984.

[3] P. Brusilovsky and T.D. Loboda. WADEIn II: a case for adaptive explanatory visualization. *ACM SIGCSE Bulletin*, 38(3):48–52, 2006.

[4] J.H. Davies. *MSP430 microcontroller basics.* Elsevier, 2008.

[5] W. Dolman. *Microcontrollers en de taal C (in Dutch).* Wim Dolman, 2010.

[6] G. Goguadze. *ActiveMath – Generation and Reuse of Interactive Exercises using Domain Reasoners and Automated Tutorial Strategies.* PhD thesis, Universität des Saarlandes, Germany, 2011.

[7] B. Heeren and J. Jeuring. Feedback services for stepwise exercises. *Science of Computer Programming*, 88:110–129, 2014.

[8] B. Heeren, J. Jeuring, and A. Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370, 2010.

[9] H. Keuning, J. Jeuring, and B. Heeren. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of Innovation and Technology in Computer Science Education*, pages 41–46, 2016.

[10] A.N. Kumar. Results from the evaluation of the effectiveness of an online tutor on expression evaluation. *SIGCSE Bull.*, 37(1):216–220, 2005.

[11] T. Olmer, B. Heeren, and J. Jeuring. Evaluating Haskell expressions in a tutoring environment. In *Proceedings of 3rd International Workshop on Trends in Functional Programming in Education, TFPIE 2014*, pages 50–66, 2014.

[12] J. Pardue. *C programming for microcontrollers.* SmileyMicros, 2005.

[13] T. Sirkiä. Exploring Expression-level Program Visualization in CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 153–157, 2014.

[14] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *Trans. Comput. Educ.*, 13(4):15:1–15:64, 2013.

[15] J. Sorva and T. Sirkiä. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 49–54, 2010.

[16] K. VanLehn. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education*, 16(3):227–265, 2006.