Can Tools Reduce the Costs of Developing Control Engineering-Based Self-adaptive Systems?

Tharindu Patikirikorala, Alan Colman, and Jun Han
Swinburne University of Technology
Victoria, Australia
{tpatikirikorala, acolman, jhan}@swin.edu.au

Abstract—Many self-adaptive techniques primarily based on control engineering methods have been proposed for different software system in the literature. These works only evaluate the self-adaptive capabilities of the proposed control solution, but no evaluation is performed to quantify the costs of implementing such a control solution. This paper provides details of an empirical study and its results, conducted to quantify the implementation, testing and knowledge requirement costs when building self-adaptive software systems using control engineering methods. Our objective is to investigate, whether these costs can be significantly reduced if the supporting tools are available. In order to achieve this objective, an empirical study is conducted with a group of software engineers. The findings indicate that the aforementioned costs can be significantly reduced when supporting tools are available. We also list the lessons learned from this study and recommendations, which may be useful in designing similar experiment and improve the validation of self-adaptive techniques in the future.

I. INTRODUCTION

Software system’s ability to self-adapt under changing requirements and environmental conditions has been identified as an important capability that will be needed in increasingly complex software applications in a number of domains [11]. As a result, a wide range of approaches have been proposed and investigated to realize such self-adaptive capabilities in various software systems. Surveys on such approaches can be found in [16], [9].

Among the approaches, control engineering methodologies have been widely adopted to implement self-adaptive capabilities, in particular for the cases where certain parameters have to be reconfigured at runtime, in order to achieve the business objectives under changing conditions and unpredictable events. This is because, as Shaw suggested in [17], if a software system faces external disturbances and events that cannot be modeled by the existing software engineering methods, then using the concepts from control engineering in the software architecture may be suitable. As a consequence, many academic research efforts have investigated suitable control engineering solutions to realize self-adaptive software systems. A survey of 161 such investigations can be found in [14]. These research works also include the approaches proposed by the research labs of HP and IBM (e.g., [4], [7], [5], [19], [12], [20], [22]). However, even with such significant efforts control engineering methodologies are not yet been applied or accepted in general software engineering practice to any significant extent.

(Brun, 2010) [2] argues that the existing works evaluate just the self-adaptive capabilities and performance aspects of the proposed solution, but no evidence has been given to identify the costs and benefits of applying such approaches. This factor has been quantitatively illustrated by [21] as well. This is equally true for the self-adaptive software systems implemented with control engineering methodologies. From the survey [14], we also observed the same shortcoming in the evaluation of all the control solutions proposed thus far. If control solutions are to be adopted in general software engineering practice then not only do they need to be technically effective, but the engineering costs of implementing, testing and integrating such approaches need to be established. Another impediment to the uptake of these approaches is that a control system design requires extensive knowledge in the control engineering discipline.

As we will show, a software engineer does not typically possess these specialized skills and knowledge in control system design and development. Consequently, the engineering costs of control engineering solutions are significantly high. By abstracting away some of the complex details, appropriate control system development tools may be able to reduce the engineering costs of control solutions by shortening the development time and reducing the amount of specialized knowledge required from a software engineer.

In this work, we present the results of an empirical evaluation that quantifies the extent to which a supporting tool can help to reduce the development costs of a selected control solution, and identify a set of issues related to knowledge requirements in building such a control solution. In order to achieve this objective, an experiment was designed and conducted with a group of software engineers who have extensive experience building production software solutions. The hypothesis of this experiment is “Can supporting tools reduce the costs of developing control solutions for software systems”. To the best of our knowledge this is the first time such an evaluation has been done. The results indicate that supporting tools can help to substantially reduce the costs involved in developing control system implementations, and the knowledge embedded in such tools can enable software engineers, who lack a background in control engineering, to implement control solutions that they would not otherwise be able to implement if they had to develop them from scratch. Although, this study does not quantify costs for all the control
solutions in the existing literature, nor it provides all the factors that have to be considered in the design of control solutions, the conclusions and lessons learned from this empirical study may be useful to assist the design of future investigations into the development costs of self-adaptive solutions based on control engineering. Furthermore, we provide a set of recommendations and directions to future research.

Section II provides the background related to this work. Sections III and IV present the details of the experiment and its results respectively. In Section V the lessons learned from this experiment will be covered followed by the list of limitations of this study in Section VI. Section VII overviews the related work before making the concluding remarks in Section VIII.

II. BACKGROUND

This section firstly provides an overview of feedback control system and its design process. Secondly, we investigate whether software engineers or software engineering students have educational background to design such control systems. Thirdly, we briefly describe an engineering process to develop a control system for software system.

Feedback control system and its development process. Figure 1 shows a block diagram of a feedback control system. The target system provides a set of performance variables referred to as measured outputs (or simply outputs). The sensor monitors the outputs of the target system, while the control inputs (or simply inputs) can be adjusted through the actuator to change the behavior of the system. The feedback controller is the decision making unit of the control system. The main objective of the controller is to maintain the outputs of the system sufficiently close to the desired values, by adjusting the inputs under disturbances and external events. These desired values are called in control system terms as the set point signals, which gives the option for the control system designer to specify the goals or values of the outputs that have to be maintained at runtime.

Fig. 1: Block diagram of feedback control system

Feedback control systems are widely adopted to control different processes in chemical, manufacturing and automotive industries [6]. In order to develop a feedback control system from starch several tasks have to be carried out. They are 1) listing the requirements and control objectives of the target system, 2) examining the target system and identifying the suitable system inputs and outputs followed by the development of actuators and sensors, 3) estimating mathematical models to represent the dynamics of the target system, 4) designing the control system and architecture, 5) testing the design with simulations and finally 6) implementation of the control system in the production system. After initial two tasks, the rest of the tasks follows a cyclic process till the completion of the project [6]. Furthermore, as mentioned by Goodwin et al. [6], the control engineer is an essential part to carry out each of these tasks. In order to assist the control engineer to carry out these tasks and to develop rapid prototypes, the computer aided control system design tools (e.g., Matlab) are available. The finalized control systems are typically implemented in the devices such as programmable logic controllers using the aforementioned tool support and integrated in to the target system subsequently.

The control engineering methods are attractive to develop self-adaptive systems because they (i) achieve the prescribed management objectives under unpredictable disturbances, typically by adjusting certain numerical parameters exposed by the system, (ii) provide systematic and formal design process to implement the management system compared to the rule-based approaches and (iii) provide management decisions with less or no human interventions, thereby reducing costs involved in managing the system [14].

However, currently there are no well-established processes or tools supporting the implementation of control systems to build self-adaptive software systems. In the literature typically all of the above mentioned tasks are carried out, but there is no information on how the proposed approach would be applied in production software systems. The existing studies proposed control solutions ranging from basic to complex schemes [14]. If we refer back to the task (3) in the above engineering process, the dynamics of different software systems have been modeled mathematically using either black-box, queuing and analytical models having different complexity levels. Similarly, from our survey [14], we also identified that different management problems have been tackled using control schemes having different complexity levels, which include Propositional Integral Derivative (PID), adaptive, model predictive, linear quadratic, hierarchical and decentralized control schemes. In addition, the application domains of these control solutions are different as well (e.g., middleware, data storage and real time systems [14]).

Control engineering knowledge of software engineers. In this segment we examine the extent to which current software curricula cover control engineering, and take this as an indication of the level of control engineering knowledge held by typical software engineers. According to the above description, to design a suitable control system, substantial knowledge and skills related to control engineering is required by the engineering team. Let us start with the assumption that a software engineer or a software engineering team has enough educational background to implement a control system in a systematic way. If this assumption holds, there is less or no impendent to build a control engineering solution for any software engineer. In order to validate this assumption we perform a hypothesis test. The null hypothesis ($H_0$) and alternative hypothesis ($H_1$) are as follows.

$H_0$ = ‘Majority of the existing software engineering courses contain subject material related to control system design’

$H_1$ = ‘Majority of the existing software engineering courses
do not contain subject material related to control system design'.

To validate the hypothesis, we have to select a list of universities conducting software engineering courses. This is not a straight-forward task. It is because (1) not all universities conduct software engineering courses, (2) there are several types of courses fall under the umbrella of software engineering (e.g., courses in computer science and engineering, information and communication engineering) and (3) quality of the conducted courses may differ from one university to another. To address the first and third reasons, we used a list of top 200 universities in the world conducting software engineering courses from (www.topuniversities.com). From that list, we sampled 30 universities randomly. In each of these sampled universities, the curriculum of all software engineering courses were downloaded and analyzed (this is to address consideration (2) above). In this analysis, we classified universities into three groups. They are universities providing a control engineering related subjects directly (there is a subject in the curriculum) or indirectly (a related software engineering subject covers it briefly e.g., embedded system subject) and universities not providing any subject related to control engineering within the software engineering courses. Table I shows the results for each of these groups.

<table>
<thead>
<tr>
<th>Group</th>
<th>Number of universities</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covered directly</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Covered indirectly</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Not covered</td>
<td>25</td>
<td>80</td>
</tr>
</tbody>
</table>

From the sample statistics it is evident that only 10% of the universities covers control system design subjects, while 90% of the universities do not provide such subjects directly. To generalize this result for the top 200 universities, we selected 95% confidence interval and computed minimum and maximum percentages. Based on this analysis we can say that with 95% confidence, the percentage of universities covering control engineering subjects are within the range of 0% to 21%. Therefore, we can accept the alternative hypothesis and reject the null hypothesis, i.e., majority of existing software engineering courses do not contain subject material related to control system design. Furthermore, most of the subjects covering control engineering material directly or indirectly are electives. As a result, we cannot make the assumption that all the students in those universities have selected the particular subject. In order to further evaluate the above hypothesis, we investigated the reports compiled by the joint task force of computing curriculum from ACM and IEEE. These reports provide detailed recommendations, guidelines and desired curriculum for the various computing-related disciplines based on the body of knowledge required (see, http://www.acm.org/education/curricula-recommendations). From these reports, information systems, information technology, and software engineering disciplines do not include any control engineering subjects, but for computer engineering courses, 'control system design' subject is included as a fourth year elective.

From the above evidence, we can conclude that majority of the software engineers will not have enough knowledge on control system design. Even if a student who has taken a short course on control engineering he/she may not have enough knowledge to tackle complex control problems in a software system. Given control engineering approaches are not well-established in the development of software systems in industry it is unlikely that software engineers would be able to gather this knowledge from their professional experience. This conclusion is supported by the fact that none of the software engineers that participated in our study (see, Table II in Section IV) had enough background knowledge to build control systems. As a result, even if they were to be able to develop a control solution it would unlikely to be either suitable or optimal.

We argue that design of a control system to implement a self-adaptive software system cannot be carried out by a software engineer or team. In order to assist the development process, involvement of a control engineer with sufficient background on control system design is essential [6].

**Type of engineering process required in developing control systems to build a self-adaptive software system.** We now discuss how both control and software engineers would need to contribute knowledge/skills to the process (described at the start of this section) of developing a control system for a software system. Figure 2, illustrates in the left hand column the typical tasks and process involved in developing a control system [6], while the right hand column summarizes who would have the knowledge to carry out the tasks. As mentioned above, firstly the organization management has to identify the requirement of designing a control system for their software systems. Secondly, the inputs and outputs of the system have to be identified, followed by the design of the actuators and sensors. The software engineers and the control engineer have to work together in this step, because the required system variables and way to access them differ from system to system. Once the open-loop system have been designed control engineer can use well-established control engineering methods (e.g., system identification [8]) to derive a mathematical model to present the behavior of the system. Afterwards, the designing, simulating and testing of the required control system have to be done. There are tools to support control engineers in such tasks (e.g., Matlab). In tasks (3) and (4), it is evident that a wide range of knowledge is required on the control engineering discipline. As shown above, typically software engineers do not have enough expertise to perform those tasks, therefore those two tasks are solely carried out by the control engineer.

Building a self-adaptive software system equipped with control systems is primarily a software engineering project, because the control system has to be deployed in the software
system environment. The fifth phase is to implement the control system using software components, i.e., to implement self-adaptive software system using well-established software engineering practices. However, control engineers typically do not have enough expertise to do this implementation following the software engineering concepts. Similarly, a software engineer doesn’t have enough expertise to engage in the previous two steps. In order to bridge this gap, the design specification document (we call it as the control architecture specification) has to be prepared by the control engineer at the end of second stage\(^2\). This specification shall include the block diagrams of the control system components, communicated data between components, concrete implementation of each component with the configuration parameters. Given the control architecture specification, at the fifth step, a software engineer can go about implementing the desired control system from scratch following the well-known software engineering principles.

In the sixth and final step, the implemented control system has to be deployed in the software system environment, followed by rigorous testing. This step has to be conducted by the control engineer in conjunction with the software engineer. After the test and validation of the control system, the final fine tuning of the configuration parameters of the control system has to be done at this stage. If the designed system fails to achieve the control objectives, the control engineer should go to the third stage again and have to redesign the control system.

This engineering process resolves some of the major issues related to knowledge gap involved in the design of a self-adaptive software system based on control engineering methods. However, there is still some friction left in this design process, in particular, when the concrete implementation is done by a software engineer. They are 1) issues for accurately understanding and implementing the complex mathematical details in the control architecture specification 2) testing and validating the correctness of the implementation because the control engineer may not be able to involve in this step.

The focus of this paper is to identify and quantify the costs involved with the fifth step of the above engineering process. That is to identify and quantify the costs when a software engineer develops the self-adaptive system based on the specification given by the control engineer. In addition, we investigate whether these costs can be reduced if the supporting tools are available. However, our investigation is tightly coupled with the assumption that the above described engineering process will be followed in the development of the particular control solution for a software system. Similar, engineering process are followed in other industry disciplines (e.g., chemical and manufacturing) as stated in [6].

III. Experiment Design

This section provides the details of the experiment designed. In Section III-A, we present the hypothesis validated by this experiment. Section III-B overviews the details of sample profile and size followed by experiment procedure and data analysis methods in Sections III-C and III-D respectively.

A. Hypothesis

In this work we formulate the following null ($H_0$) and alternative ($H_1$) hypothesis.

$H_0$ = “Supporting tools can not reduce the costs of developing control solutions for a software system”

$H_1$ = “Supporting tools can reduce the costs of developing control solutions for a software system”

Here, we assume that the engineering process in Figure 2 is followed and the control architecture specification is available to the software engineer. Then, the objective of this hypothesis is given the tool support, a software engineer can implement the desired control system correctly with less costs compared to the case of implementing the control system from scratch. The major costs involved in such a development are the implementation and testing costs and knowledge required in the implementation. The implementation and testing costs of a control system can be measured based on the time taken and number of source lines of codes (SLoCs) required to complete the given control system development task. These parameters will be used to statistically validate the hypothesis. In contrast, the knowledge requirement cost is hard to measure, therefore will not be used to validate the hypothesis. More details on how we draw conclusions on the knowledge requirement will be provided in Section III-D.

B. Sample Profile and Size

The target population for this experiment is software engineers. It is impractical to conduct an experiment involving a large number of software engineers, therefore sampling (selecting representative set of participants) from the population

---

\(^2\)This concept is similar to software requirement specification in general software engineering process.
is the typical approach [1]. We set out following criteria when sampling the participants for this experiment to improve the quality of the study.

1) A participant must have at least a degree related to computer engineering, software engineering or information technology - this is to maintain the level of education background.

2) A participant should have more than 2 years experience in developing production software systems - this criteria provides us the confidence that each participant has sufficient industrial experience in software engineering.

3) A participant should have experience in Java or .NET platforms - the experiment tools are only compatible with these two languages. Consequently, knowledge on either one of those platforms is an important requirement.

The experiment procedure is an expensive process (see, Section III-C), therefore finding participants was difficult. As a result, we opted to use convenience sampling instead of random sampling. The convenience sampling [1] is one of the common sampling techniques and more practical method in real world compared to random sampling [10]. Based on author’s contacts, several organizations and software engineers were contacted (via phone and email) to find participants for this experiment. The sample size is an important aspect that affects the generalizability of the results of a quantitative research [1]. The sample size has to be selected based on the desired statistical power of significance and population size. The designed experiment approximately takes around 6 to 7 hours. Therefore, a substantial amount of time and human resources are required to conduct an experiment involving a large number of participants. As a result, we settled for a smaller sample size and then to use non-parametric statistical test (e.g., Mann-Whitney test) [18] compared to using parametric tests, which require relatively larger sample size. However, a limitation of non-parametric statistical tests is the power of significance is low compared to parametric statistical test.

From the people authors approached, 14 software engineers volunteered to participate in this experiment. The information of these participants is listed in Table II.

C. Procedure

This is the main stage of the experiment design. We faced many challenges at this stage, because there are no experiments designed to evaluate cost of engineering self-adaptive systems so far in the literature. The objective of the experiment procedure is validating the hypothesis and then to identify and quantify costs of developing control systems and whether tools can support software engineers to reduce these costs. The major questions we faced were what is the development task that should be given? and what should be the tool support provided to the software engineer?

Development task. Many different control schemes have been proposed in the existing literature at different complexity levels as mentioned in Section II. Following characteristics were considered in the selection of the development task, 1) the ability to complete it in short time and 2) have a sufficient complexity to draw conclusions on the knowledge requirements. For example, to develop a simple PID controller takes very short time compared to the development of mathematically complex model predictive or adaptive controller, which would take multiple days. In this experiment we opted to use implementation of a moderately complex multi-model switching and tuning adaptive (type 2) control system (MMSTT2) as the development task. This scheme was proposed to design self-adaptive systems based on control engineering in our previous publications [13]. Figure 3 shows the architecture of the MMSTT2 scheme selected as the development task.

The MMSTT2 scheme operates as follows. Given the input \( u \) and output \( y \) data from the target system, two models predicts the output \( \hat{y} \) for the current sample \( t \). The switching algorithm evaluates the predicted \( \hat{y} \) of each model against the actual system output. From this evaluation the model that fits the current environment and system conditions will be selected and the controller corresponding to that model will be connected to the system to make decisions till the next evaluation of the switching algorithm.

Hence, this implementation involves the development of the models, PID controllers and a switching algorithm, which only require understanding of basic arithmetic and discrete time mathematics. Based on our previous experience, this task can be completed within a short time (approximately 2 to 3 hours) as oppose to other control schemes which would require implementation of complex matrix operations and mathematics (e.g., predictive and adaptive control). It is noteworthy that, this task also includes the implementation of a basic PID scheme enabling us to identify issues related to such schemes.

Using these settings, we prepared the control architecture specification. A summary of the specification is shown in Appendix A. Since, there is no standard for the structure and contents of the control architecture specification, our objective was to provide a comprehensive description with examples in order to explain the requirement as clearly as possible.

Supporting Tool. In the existing literature there are no
tools supporting the development tasks of control solutions for a software system. Due to this fact, we strengthened the supporting tools presented in our earlier works [13], [15] and used it in this experiment. This supporting tool is a class library, which provides a rich set of standard control components and algorithms that can be effectively used in the implementations of self-adaptive software systems. The current implementation provides the following off-the-shelf standard control components.

This class library is currently available in Java and C#.Net with standard documentations.

This tool can be used to complete the above task described in the control architecture specification. The level of support provided by this class library for the development task is the configurable components with the implementations of First order models, PID controllers and MMSTT2 scheme together with the standard documentation describing the details of each component. These components are standard classes which encapsulates the algorithms and equations of corresponding components. Given the required input, output data and turning parameters for these components, the desired outcomes can be achieved by executing a function call. In order to successfully complete the task using this class library, the software engineer has to read the documentation and understand the functionalities of the required components and then has to enable inter-communications between them correctly.

Participant and task grouping. There are two ways to group the participants and tasks. First, we can divide the participants into two groups, where one group completes the task with the tool support while the other group completes it without any tool support. However, due to the limited amount of participants maintaining similar quality attributes (such as the professional experience, ability to understand the development task, speeds of coding) between groups is difficult. This fact could lead to significant bias in the experiment results. The second design option is to let each participant complete the task in both ways, i.e. without the tool support (T1) and with the tool support (T2). This option reduces the issues related to the first option, but raises bias of learning effect due to the order of the tasks being carried out (i.e., T1 after T2 or T2 after T1). The learning effect would allow a participant to complete the second task in a quicker time than they would have done without prior experience gained on the problem and solution when completing the first task. For this experiment design we used the second option. To mitigate the issues of learning effect, the order in which these tasks were undertaken was randomly selected between participants.

Experiment process. Now, we move on to the details of how the experiment was conducted with each participant. The experiment has four stages. In the first stage each participant received a control architecture specification describing the control system to be implemented. They then participated in an oral presentation conducted by a control engineer, which describes the details of the control architecture specification. This is followed by a question and answer session where each participant can clear any doubts before going to concrete implementation.

The second stage was to prepare the workstation of the participant with the experiment instrumentation. All the participants used the workstation they use to do their day to day software development. The instrumentation included following entities.

1) A simulation of a target software system with a sensor and actuator. This component generates artificial input and output data from a target system, which is required by the MMSTT2 control system.
2) A graphing tool and test case. The graphing tool enables to compare the output of the implemented control system against the test case or desired output.
3) Class library with the documentation. The supporting tool that has to be used when conducting T2.
4) A data sheet. To enter the start time and end time of T1, T2 and other related information of the participant (name, experience, knowledge on control engineering).

All participants were educated about the instrumentation and how to use them during the implementation. In addition, they were informed about the order in which the tasks have to be carried out. That is whether to complete the assigned implementation in the control architecture specification from scratch (T1) first and then using the class library (T2) or vice versa. Each implementation completes when the test case is successful.

In the third stage each participant carries out both tasks. During this stage a control engineer was available to ask any question regarding control engineering or control architecture specification. These questions asked by participants were collected by authors in a transcript during the experiment for qualitative analysis. If the question is related to the class library, no additional information is given to the participant, apart from the standard documentation.

The final stage is to gather data and conduct the analysis.

D. Data Collection and Analysis

The following items were collected after completion of each experiment.

1) Full data sheet.
2) Source code files used to complete T1 and T2 separately, and
3) Transcripts of the questions asked.

In the data analysis stage, the main objective is to gather evidence to accept or reject the hypothesis formulated in Section III-A. For this we have to compile quantitative data. To quantify implementation and testing costs, two properties were measured. First, from the data sheets, the time taken to complete T1 and T2 were calculated (this includes testing


This is to avoid any bias if they had to develop it in a new workstation that they are not familiar with.
cost as well). The second property is the SLoCs required to complete T1 and T2. This is a measure of implementation effort [3]. In order to calculate the SLoCs in a standard way we used the locmetrics\(^5\) tool. Then, to draw conclusions on the knowledge requirement costs for the particular control system implementation, we also analyzed the questions in the transcripts. Generating quantitative data from the qualitative data is a challenging task. A content analysis was conducted to generate quantitative results from the transcripts of questions. We categorized the questions related to the component (model, controller, switching scheme) initially and then did a frequency analysis to quantify the occurrences of the same question. Afterwards, the results of the frequency analysis were used to draw conclusions on the knowledge requirement costs.

IV. RESULTS

Table II\(^6\) summarizes the findings of the experiment, which include the time and SLoCs of T1 and T2 for each participant. The prominent observation is the time and SLoCs taken to complete T2 (using the library) is low compared to T1, irrespective of the order in which the tasks were carried out. However, the SLoCs required for T1 varies drastically in contrast to T2. The analysis of the code indicated that SLoCs were increased (e.g., P2) because the implementation was extendable and configurable compared to other implementation which lack those characteristics (e.g., P3) for the case of T1. The variations of SLoCs are less for T2 because when class library is used, the successful implementations show a similar pattern. The time consumed for both T1 and T2 varies a lot from participant to participant. In the case of T1 time varied mainly because of the time taken to debug and validate the implementation. The reason to have variable time in the case of T2 is because of the time taken to read and understand the documentation of the tool.

In summary, on average (1 hour and 28 minutes) of time and (137 SLoCs) have been reduced when the supporting tool is used compared to implementing the entire control system from scratch. Using these quantitative results we also performed a power analysis using Mann-Whitney test [18] to compare the significance of the results. For the case of time and SLoCs the p-values generated from the Mann-Whitney test are (7.4250e-006) and (7.4037e-006) respectively. These p-values are smaller than the significance level of 0.01, which means that time and SLoCs data of T1 and T2 are significantly different. Subsequently, by comparing the average statistics from Table II, we can reject $H_0$ and accept $H_1$ (see Section III-A). That means tools can reduce the costs of developing control systems to build self-adaptive software systems.

In addition, the content and frequency analysis reviled that cost on knowledge requirement is high for the case of T1. When participants were carrying out T1, the control engineer had to engage in the debugging and testing stages of the implementation by answering many questions till the test case was successful. These questions were analyzed using the process explained in Section III-D. Some of the questions with high frequencies are summarized in Table III.

In the case where the library was used (T2), none of the above problems were observed in relation to the mathematical background of the implementation. Most questions were related to some of the mismatches and ambiguity in the standard control engineering terms used in the control architecture specification and the documentation of the class library. However, the documentation provided sufficient information in this implementation for most participants.

Although the control system that the participants had to implement in this experiment was at a moderate level of complexity, the findings of this experiment indicate the use of the library reduced the costs related implementation time, effort and knowledge requirements. For schemes at higher level of complexity and with rigorous mathematical foundation (e.g., adaptive and predictive control), we can speculate that even greater benefits could be gained by using the supporting tools.

Table II\(^6\) summarizes the findings of the experiment, which include the time and SLoCs of T1 and T2 for each participant. The prominent observation is the time and SLoCs taken to complete T2 (using the library) is low compared to T1, irrespective of the order in which the tasks were carried out. However, the SLoCs required for T1 varies drastically in contrast to T2. The analysis of the code indicated that SLoCs were increased (e.g., P2) because the implementation was extendable and configurable compared to other implementation which lack those characteristics (e.g., P3) for the case of T1. The variations of SLoCs are less for T2 because when class library is used, the successful implementations show a similar pattern. The time consumed for both T1 and T2 varies a lot from participant to participant. In the case of T1 time varied mainly because of the time taken to debug and validate the implementation. The reason to have variable time in the case of T2 is because of the time taken to read and understand the documentation of the tool.

Table III: Frequency analysis of the questions

<table>
<thead>
<tr>
<th>Id</th>
<th>Question</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controller</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Do we need the previous error ($e(t-1)$) of equation (2)) for PI controller?</td>
<td>29</td>
</tr>
<tr>
<td><strong>Model</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Which time sample values to use in the model (see equation (1))?</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>What is the difference between predicted output ($g_p(t)$) vs original system output ($g(t)$)?</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>What are the system input, output and controller input and output?</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Is prediction error ($pe(t)$) and control error ($e(t)$) same?</td>
<td>21</td>
</tr>
<tr>
<td><strong>Switching algorithm</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>How to implement the switching period (see equation (3))?</td>
<td>43</td>
</tr>
</tbody>
</table>
V. LESSONS LEARNED, RECOMMENDATIONS AND DIRECTION FOR FUTURE RESEARCH

When designing the experiment many challenges were encountered. It is because to the best of our knowledge this is the first experiment that quantifies the relative costs of developing self-adaptive software based on control engineering methods. In particular, deciding on the contents of the control architecture specification, selection of the control scheme to be implemented, deciding on the level of support given by the tool and deciding on task ordering were major decisions we had to make. In this section we list lessons learned from this experiment and provide some recommendations for similar experiments in the future.

If the control architecture specification is intended for an implementation without any tool support, the mathematical formulations related to the implementation have to be described in depth. From the implementation of MMSTT2 scheme, we observed difficulties of 1) thinking relative to sample time or in other words lack of understanding on discrete time mathematics (see questions (1), (2) and (6) in Table III), 2) differentiating the system output, input, the controller input, output and the model prediction and the original system output (see questions (3), (4) and (5) in Table III) for some participants.

Hence, emphasis on the discrete time mathematics and clear definitions of related variables in the specification is important. Each component in the architecture has to be explained with input and desired output variables of the component, their data types, initial values, whether they are constants or configurable in order to improve the quality of the designed component. Furthermore, to assist the validation of each component, it is useful to including the test cases and examples in the specification.

If the control architecture specification is intended for an implementation with a supporting tool, using the standard control engineering terminology and equations in the document as well as using them in the tool and its documentation is paramount to avoid confusion. Extensive details of each component are not required, but values for all configuration parameters have to be specified in the specification.

The selection of the task largely affects the outcome of the experiment, especially, for the case of drawing conclusions on the knowledge requirements. A complex scheme (e.g., adaptive, hierarchical) may give us more evidence related to the knowledge requirements. However, participants may give up the implementation if the sufficient knowledge is not available, which might be a risk in such complex implementations. Furthermore, implementation time complexity is another concern. Consequently, the task has to be selected carefully.

We opted to encapsulate mathematical complexity and algorithms inside the components and provide a class library as the supporting tool. As evident from the outcomes of the experiment, this approach is successful. We noticed that task with the tool support was carried out more efficiently by most of the participants, because the use of 3rd party class libraries and tools is a standard practice in software development. As a result, most participants knew how to navigate and find out related material from the documentation of the class library. However, clear definitions of each parameter, function methods and details of the control algorithm implemented in each component and sample codes of how to use each component are also important for rapid development.

In this experiment all participants completed the task using the tool and without using the tool. Although we changed the order of how tasks were undertaken to reduce the impact of leaning, there were several drawbacks in this approach. When the T2 follows T1, some participants tried to follow their design in T1 to complete T2 as well. In this case, the impact on T2 is less because a certain pattern has to be followed based on the implementation of the class library to complete T2. Similarly, when T1 follows T2, some participants tend to have high-level design of the class library in their design of T1. In this case, the internal logic of all components has to be implemented by each participant from scratch, consequently there is less effect on the outcomes of T1. However, these concerns lead to some bias in the results. To reduce this bias, the experiment can also be designed with two sets of participants each doing only one task as mentioned above. In this case, a substantial number of participants have to be grouped in each set to maintain the quality attributes (e.g., professional experience, habits, coding speed and mathematical knowledge) of each set similar.

Although from the above results and discussions we have shown that the tools can help mitigate the engineering and knowledge costs related to control solution development for software systems, there have to be further investigations into 1) what are the best methods with lowest cost to design experiments to evaluate engineering costs and 2) what are the characteristics of the tools to provide best support to a software engineer. For example, related to the first point, how to decide 1) a suitable task, 2) task ordering, and 3) group formation in the experiment has to be further investigated. A couple of questions related to second point are can just a class library with simple and complex control component is enough to engineer all types of control solutions, or should the tool support the layout of complex control system architecture on top of the existing software system architecture as well.

VI. LIMITATIONS OF THE STUDY AND TREATS TO VALIDITY

There are several limitations in this study. Although the experiment was based on a typical control engineering process as presented in Section II, in practice other process might be adopted for the development of control systems to build self-adaptive software systems. In addition, there is no standard way to write a control architecture specification. To mitigate this issue, 1) we explained all components and mathematical details as clear as possible giving example as well, 2) an oral presentation on the implementation task was given by the control engineer and 3) a control engineer was available to
ask any questions during the implementation stage. However, formulation of control architecture specification for software systems is an open problem, which needs further investigation. The results presented from this study are highly dependent on the control scheme selected as the development task. If only a simple control system such as PID controller was selected to be developed the relative cost saving of T2 over T1 would be much less. In contrast, if a more complex scheme such as adaptive, predictive and MMSTT type 4 schemes was selected, the relative costs of T1 is likely to be significantly greater than T2.

The sampling was done using convenience-sampling technique compared to random sampling. This may include some bias in participant selection. Furthermore, there are two issues to generalize these results. Firstly, since the sample size is small, we opted to use non-parametric statistical test, which has lower power compared to parametric tests. Secondly, the results of content analysis related to the knowledge requirement are not a comprehensive list of knowledge requirements to implement all possible control systems. The list shown in Table III is only valid for the case of MMSTT2 scheme.

VII. RELATED WORK

To the best of our knowledge, only one study exists in the literature quantifying the development cost of self-adaptive systems. Cheng et al. [3] compare the engineering cost of a self-adaptive system based on their proposed approach against the engineering cost of self-adaptive system without their approach using a single participant. This study is complementary to our study. However, their implementation task is based on software engineering only and has no relation to control engineering. Although there are many control engineering approaches proposed thus far in the literature (see our survey [14]), none of them have evaluated the engineering costs of such solutions.

In the existing works, the general purpose software such as Matlab\(^7\) has been used to implement proof-of-concept systems. Some works have connected Matlab via pipelines. Such techniques are suboptimal for production software environments, which could lead to security hazards and performance bottlenecks. In other industry disciplines there are computer aided control system design tools, which enable control engineers to prototype control system fast. However, the deployment support of these tools limits to programmable logic controllers or microcontrollers, which means they do not support laying out complex control solutions on top of a software system architecture or their special and characteristics.

In [4] an approach is described to automate the design of a control loop by encapsulating the control engineer’s expertise into several software agents. Depending on the complexity of the requirements and system characteristics such an approach is difficult to apply without a human expert in the design process of even a simple control system.

VIII. CONCLUSION

Although there are many self-adaptive approaches based on control engineering methods, there is no empirical evidence quantifying the engineering costs of these control solutions related to the implementation effort and knowledge required. This paper overviews an empirical study conducted to quantify the engineering costs and how these costs can be reduced when the supporting tools are available for control system implementations. The objective of the paper was to validate the hypothesis “Can supporting tools reduce the costs of developing self-adaptive systems based on control engineering methods?”. An experiment has been designed and conducted with a group of experienced software engineers has shown that the supporting tools can help to reduce the costs involved in control system implementations significantly. In addition, we have also listed the lessons learned from this experiment and made some recommendations for future experiment designs in this area.

ACKNOWLEDGMENT

Authors would like to thank all the participants and their respective organizations for participating in this experiment. We would also like to acknowledge Mr. Ayman Amin for assisting us with statistical tests.

REFERENCES


\(^7\)http://www.mathworks.com/products/matlab/
APPENDIX

This appendix provides a summary of control architecture specification used in the experiment.

Model
Figure 4a shows the model component. Given the sensor data (yt) and current input (ut) at the t - 1 time instance, this model predicts the output for the current time sample (yt(t)) using a mathematical equation. For this implementation autoregressive exogenous input (ARX) model is used.

Equation of first order ARX model:
\[ \hat{y}(t) + ay(t - 1) = bu(t - 1) \]  

Controller
Figure 4b illustrates the controller as a component. When sensor data (yt(t)), input (ut(t - 1)) and the objective value (set point r) is given to the controller as the input at the time t, it calculates the input for the current time sample using a mathematical equation. In this implementation velocity form of proportional integral controller is used. Equation is as follows.
\[ u(k) = u(t - 1) + (K_p + K_i)e(t) - K_pe(t - 1) \]  

Error at the t time sample is calculated using \( e(t) = r - y(t) \).

Switching algorithm
Figure 4c shows the switching component. Inputs are the model predictions and the system output for the current time sample.

Prediction error: \( pe_i(t) = \hat{y}_i(t) - y(t) \) where, \( \hat{y}_i(t) \) prediction of ith model ie A, B. \( y(t) \) is the output of the system. For each model, the prediction error has to be calculated. i.e., \( pe_A(t) \) and \( pe_B(t) \).

Say k is the current time instance, \( \alpha, \beta, T \) are given. The following algorithm is executed every Tth time sample. There are two main steps.

Model evaluation:
\[ J_i(t) = \alpha pe^2(t) + \beta \sum_{r=0}^{t} pe^2(r) \quad i \in \{A, B\} \]  

T specifies the time window. So the perditions generated by the models (A, B) for \( t - T \) time samples has to be stored and accumulated inside the switching component. Then this equation has to be used to compute the J value for each model.

Model selection: The model that generated minimum J value is selected
\[ J_{min}(t) = \min\{J_i(t)\} \quad i \in \{A, B\} \]

In every \( t + T \) sample model that generated the minimum J value will be selected and the corresponding controller will be connected to the system.

![Diagram of components of MMSTT2](image-url)