Investigating the Effects of Integrating Handcrafted Code in Model-Driven Engineering

Tim Bolender, Bernhard Rumpe, Andreas Wortmann

Software Engineering
RWTH Aachen University
Aachen, Germany
www.se-rwth.de

***Abstract*—Where model-driven engineering (MDE) requires models to interact with general-purpose programming language (GPL) artifacts, sophisticated patterns for the integration of generated and handcrafted code are required. This raises a gap between modeling and programming which requires developers to switch between both activities and their contexts. Action languages that enable interacting with GPL artifacts on model level can alleviate the need for switching, but cannot rely on the mature development tools of GPLs. We propose to investigate the acceptance and effects of MDE with action languages that GPL interaction. To this effect, we present a study design for comparing MDE with integration of handcrafted artifacts to pervasive MDE with action languages. In this study, participants develop the behavior of selected components of the software architecture of a small autonomous robot using either the delegation pattern or a Java-like action language. With this, we aim to uncover which method yields better acceptance and performance results.**

I. Introduction

Programming raises a conceptual gap between the problem domains and the solution domains of discourse [1]. Model-driven engineering (MDE) aims at reducing this gap by lifting more abstract models to primary development artifacts [2]. Such models are better suited to analysis, communication, documentation, and transformation. However, where models require interaction with general-purpose programming language (GPL) artifacts, such as legacy code, libraries, or frameworks, modeling infrastructures usually require bridging this gap by integrating handcrafted with generated artifacts. For this, various patterns have been developed [3], all of which require the modeler to switch from more abstract modeling activities to very technology-specific programming activities, which require different tooling and different mindsets. Where developers switch between various activities 47 times per hour on average already [4], the additional activity switching required by such MDE might ultimately hinder development.

We aim to investigate the effect of pervasive MDE using action languages in a comparative modeling study. In this study, the participants develop the behavior of MontiArcAutomaton [5] component models of a small autonomous robot. To this end, they either integrate handcrafted code through a variant of the delegation pattern [3] or leverage a Java-like action language. We provide the software architecture without component behavior to the participants as well as a code generator that translates the software architectures to Java code executable with the Simbad[1] 3D robot simulator. The treatment group develops the components' behavior with embedded, restricted Java/P, which is the action language of the UML/P [6] language family. The control group develops the components' behavior through integrating of handcrafted Java code. We record the 100-minute development sessions and instrument the development environment with analyses on the quality of the component behavior under development. Together with pre-study questionnaires and post-study questionnaires, we aim to uncover whether pervasive MDE with action languages or programming of component behavior method yields better acceptance and performance results.

In the following, Sec. II introduces the MontiArcAutomaton architecture modeling infrastructure and the Java/P action language, before Sec. III details the study design and **??** discusses threats to its validity. Then Sec. V describes preliminary results from a pilot study andSec. VI debates related studies. Sec. VII concludes.

II. Preliminaries

To prevent architecture modelers from switching between modeling and programming, we embed the Java/P [6] action language into components of the MontiArcAutomaton [5] architecture description language. This section introduces both.

A. The Java/P Action Language

Java/P [6] is a MontiCore [7] language resembling Java 1.7. It is used as action language in the UML/P [6] language family and supports the full concrete syntax and abstract syntax of Java 1.7 as well as its context condition rules. MontiCore's code generation framework translates Java/P models into Java artifacts. Reifying a Java as a modeling language enables to easily extend it with new modeling elements (for instance, the notions of components as in ArchJava [8]), context conditions rules (such as preventing assignment of `null` values), and

[1]Simbad website: http://simbad.sourceforge.net/

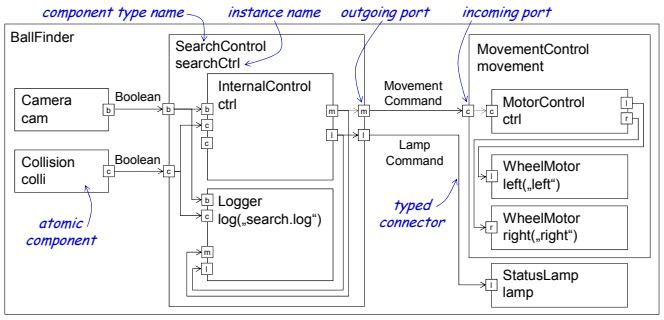

Figure 1. MontiArcAutomaton architecture of a small autonomous robot capable of finding and retrieving balls in a simulated environment.

transformations (such as automatically creating getters and setters). Moreover, MontiCore produces a parser for Java/P programs that supports processing Java 1.7 classes, *i.e.,* lifting legacy level to model level. This enables to *model* Java programs capable of using existing libraries and frameworks.

B. The MontiArcAutomaton Modeling Infrastructure

MontiArcAutomaton [5] is an extensible architecture modeling infrastructure. It comprises a component & connector (C&C) architecture description language (ADL) and a code generation framework. The ADL is realized as a MontiCore [7] language and uses MontiCore's language integration mechanisms [9] to enable plug-and-play embedding of action languages. It has been configured with behavior modeling languages including Java/P and successfully deployed to teaching [10] as well as to service robotics applications [11]. The combination of MontiArcAutomaton with Java/P is denoted AJava (for "Architectural Java").

Core concepts of the MontiArcAutomaton ADL are illustrated with the experiment's software architecture depicted in Fig. 1: the MontiArcAutomaton ADL distinguishes component types (denoted components) and their instances (denoted subcomponents). Components feature parameters (similar to constructors in object-oriented GPLs) and interfaces of typed, directed ports. They either are composed (`SearchControl`) or atomic (`Collision`). Composed components yield configurations of subcomponents that exchange messages via unidirectional connectors between their ports. Atomic components yield a behavior description in form of an embedded action language or via integration of handcrafted GPL artifacts.

The `BallFinder` software architecture comprises ten subcomponents of nine different component types. The two components depicted on its left wrap sensing functions to localize a ball in the environment and to detect possible collisions. Their messages are fed into an instance of the composed component type `SearchControl`, which uses these to determine the next navigation actions and logs both incoming and outgoing messages. It sends motor commands to another composed component taking care of navigation as well as to an instance of `StatusLamp` that serves to convey messages to observers.

III. Study Design

There are various means to achieve pervasive MDE using different language combinations as well as various patterns to integrate handcrafted code (*cf.* [3]) and measuring their differences requires concretizing the research question to comparable implementations. We use AJava as representative of pervasive MDE as both ADLs and action languages are common modeling techniques. For the integration of handcrafted code, we selected the delegation pattern [12] as representative, which has been employed in various modeling infrastructures. This section describes the design of our study based on these representatives.

A. Research Goals

When planning an experiment, it is crucial to measure only the properties from which the researchers can correctly draw conclusions. To answer our research question systematically, we applied the Goal-Question-Metric (GQM) [13] method. Using this top-down approach, the conceptual level (goals) is considered first. This is refined to the operational level (questions) from which the quantitative level (metrics) is observed. Each refinement step is based on the preceding level. This ensures that only metrics are employed, which help to achieve the original goal and not because they are easy to measure or convenient for the researcher. To this effect, our research goal is to *analyze behavior modeling with AJava and the delegation pattern for the purpose of evaluation with respect to their effectiveness and efficiency from the point of view of the researcher in the context of graduate and undergraduate students at RWTH Aachen University.*

We concretize this in three research questions as follows:

RQ1 Does pervasive MDE with AJava help to reduce the development time compared to describing component behavior with Java using the delegation pattern?

RQ2 Is the pervasive MDE with AJava less error-prone than employing the delegation pattern for integration of handcrafted code?

RQ3 How convenient is pervasive MDE with AJava for the developers?

Based on these research questions, we determine the variables to measure during the experiment (Section III-B) as well as to formulate our hypotheses to be tested against the collected data in (Section III-C). Thereby, we strictly follow the GQM method, which we use as guideline for the study design.

B. Variables

Before devising the corresponding metrics to answer our research questions, we address the variables of the experiment.

1) Independent Variables: As we stated before, we intend to gain insights into assets and drawbacks of pervasive MDE compared with the integration of handcrafted artifacts. Therefore, we have only one independent variable, which is:

- **Development method** The method used for behavior integration. This variable features two alternatives, namely

the *delegation pattern* and *AJava* as we select these as representatives for the methods in question.

Thus, we are conducting a single factor experiment with two alternatives. To this effect, we employ two test groups, one for each treatment method. To determine the composition of these groups, we employ randomization to ensure a balanced distribution with respect to, *e.g.,* knowledge and experience between the groups.

2) Controlled Variables: To be able to make reliable assertions after the experiment, we try to control the following variables:

- **Experience** MontiArcAutomaton is developed by the chair of Software Engineering at the RWTH Aachen University and part of its teaching activities. Hence knowledge about its basic functionalities and its methods can be assumed. However, we provide an introduction to ensure that all subjects participate with the same knowledge.
- **Programming skills** These are important for development in general. We expect these to be homogeneously distributed, thus we consider this variable as controlled. By employing questionnaires, we aim to ensure that our assumption on Java programming skills holds.
- **Project** The environment in which techniques are exerted has a major influence on the results. We prepare a project consisting of multiple tasks and provide a project skeleton to the participants. In Section III-E, we give an overview over this project and introduce the tasks.
- **Tooling** Tooling, such as IDEs and build systems, can reduce the work for certain tasks enormously. We ensure that the tooling support for applying both methods is identical.

3) Response Variables: The last step of the GQM method is to consider each of the research question individually and devise a list of metrics to answer these.

RQ1 Does pervasive MDE with AJava help to reduce the development time compared to modeling component behavior with Java using the delegation pattern?

Time of solution finding The time needed to resolve the whole task, including testing and error-solving.
Time for programming The time required to develop the actual solution. Includes actual programming time only, *i.e.,* excludes testing and code generation times.
Number of code generations The number of times the code generator was started.
Time for testing The time needed to test the solution.
Time needed for error-solving The time used to solve errors reported during compiling and testing.

RQ2 Is the pervasive MDE with AJava less error-prone than employing the delegation pattern for integration of handcrafted code?

Absolute number of errors The number of error occurring during compilation and testing.
Time needed for error-solving The time used to solve any errors that occurred.
Errors in Time The number of errors in relation to the time used to come up with a solution.

RQ3 How convenient is pervasive MDE with AJava for the developers?

Time saving The opinion whether the techniques is considered as time saving.
Convenience The evaluation whether the technique is found to be easy to use.
Overhead creation The assessment whether the technique requires efforts which are unnecessary for the actual solution of the problem.
Demand for more tooling The rating whether the technique needs more tooling to be used properly.

Each of these variables is measured for each of the tasks individually and once for the whole project. This way, we enable a result analysis with more insights about the specific strong and weak aspects of the methods.

C. Hypotheses

To enable the statistical analysis of the experiments results, we state our hypotheses pairs in the following. For the sake of simplicity, we give a combined expectation about each research question.

RQ1 We expect using AJava requires less development time and code generation iterations, *i.e.,*

H1.1$_0$ Solution finding time is larger or equal.
H1.1$_1$ Solution finding time is less.
H1.2$_0$ Solution creation time is larger or equal.
H1.2$_1$ Solution creation time is less.
H1.3$_0$ Code generation is run more or equally often.
H1.3$_1$ Code generation is run less often.
H1.4$_0$ Testing time is larger or equal.
H1.4$_1$ Testing time is less.
H1.5$_0$ Error-solving time is larger or equal.
H1.5$_1$ Error-solving time is less.

RQ2 We expect using AJava produces less errors and consequently requires less time for error solving, *i.e.,*

H2.1$_0$ The absolute number of errors is larger or equal.
H2.1$_1$ The absolute number of errors is less.
H2.2$_0$ The time for error-solving is larger or equal.
H2.2$_1$ The time for error-solving is less.
H2.3$_0$ The errors per time is larger or equal.
H2.3$_1$ The errors per time is less.

RQ3 We expect using AJava better accepted and conceived more comfortable, *i.e.,*

H3.1$_0$ The valuation of time saving is less or equal.
H3.1$_1$ The valuation of time saving is larger.
H3.2$_0$ The valuation of convenience is less or equal.
H3.2$_1$ The valuation of convenience is larger.
H3.3$_0$ The valuation of overhead is larger or equal.
H3.3$_1$ The valuation of overhead is less.
H3.4$_0$ The demand for tooling is larger or equal.
H3.4$_1$ The demand for tooling is less.

As the variables are collected for each task individually, the hypotheses are validated respectively. We employ a one-sided Student's t-test for those metrics that are of a ratio scale, *e.g.,* development time or error count. For the metrics, especially those of **RQ3**, we use a one-sided Mann-Whitney-U test since we deal with measured data of ordinal scale.

D. Participants

Our experiment represents an internal validation, *i.e.,* the participants are computer science students at the chair of Software Engineering. This includes Bachelor, Master, and Ph.D. students, all of which expected to have a solid knowledge in Java due to their curricula and background as well as interests in software architecture. Furthermore, we anticipate that a majority of the participants already is aware MontiArcAutomaton and has a rudimentary idea of its capabilities due to its inclusion in the the department's teaching and research activities. Nevertheless, an introduction to MontiArcAutomaton is given to ensure a consistent foundation for all participants. Based on the in the experiment included questionnaires, a more detailed description of the demography of the participants including their prior knowledge can later be given during the result analysis. We expect to have 20 subjects participate in the experiment.

E. Experiment Materials

During the experiment, the participants have to complete a series of separate tasks in a predefined order. Each of these tasks has the goal to implement a particular component behavior in an overall project, but is itself independent. For each of these components, an exact description of the behavior is provided in form of an implementation-independent text description. Since we are only interested in the behavior implementation during the experiment, an architecture skeleton of this project will be provided.

Each participant is implementing the software for a robot on the basis of the open source Simbad robot simulator framework [14]. Simbad features a 3D virtual environment in which a virtual robot can be executed. The robot can be equipped with a variety of utilities, which enable it to sense objects and obstacles which can be placed in this environment. The movement is realized via two different kinematic models to choose from (1) the default composite control with a velocity and a rotation value; and (2) the differential control, in which two wheels are separately controllable. The individual behavior is implemented in a dedicated method which is repeatedly called by the simulator framework. To enables the supervision of this behavior, Simbad features a simple interface which renders the current as state of the environment as well the different sensor values. Exemplary, the user interface with the experiment's scenario is depicted in Fig. 2. It enables the control over the simulation, and provides a 3D environment visualization as well as details about the simulation and the robot state (left).

The robot software architecture implemented by the participants during the experiment is called `BallFinder`. As the name suggests, its task is to find and collect red balls.

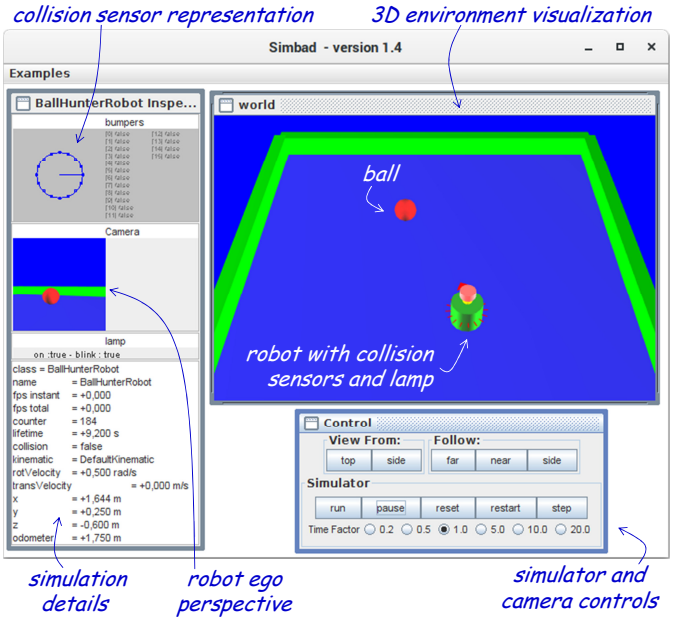

Figure 2. The graphical interface of the Simbad robot simulator.

Table I
TASK COMPONENTS WITH THEIR IMPLEMENTATION CHALLENGES

Component Name	Logic	Integration	Refactoring
Camera	✓	✓	
Collision	✓	✓	
Logger	✓	✓	
MovementControl			✓
MotorControl	✓		✓
CompositeMotor		✓	
WheelMotor		✓	

Therefore, it is equipped with a software camera for locating balls and a range of bumper sensors for collision detection. To provide easy visual feedback to the observer, it features a simple lamp as actuator. The search takes place in two separate phases: locating and collecting. During locating, the robot rotates around its axis until a red ball is recognized through a rudimentary image analysis via its camera. When successful, the robot switches into the collecting phase: it stops rotating and starts moving forward towards the ball until a collision is detected. During the first phase, the lamp is supposed to blink while during the second it should be turned on continuously. After a successful retrieval, a new ball appears at a different spot in the environment and the whole procedure starts all over again.

The architecture used for this robot was already introduced by Fig. 1. Almost each of its components is an experimental item to which the two treatments should be applied to. Therefore, the subjects should develop the components' the behavior implementations. Additionally, one task requires to refactor the architecture to change the robot's kinematic model. In Table I, an overview of the components to be implemented is given

with the kind of their related implementation challenges. We distinguish between three main challenge kinds:

1) **Logic** Tasks of this kind demand a behavior implementation with a minimal degree of logic. This should involve the usage of control structures like branches and loops. Thus, we expect the subject to implement more than simple variable reading and writing.
2) **Integration** This kind includes the interaction with a native API. Instead of developing self-contained code, external code in form of provided functions or classes are employed. Therefore, the subject has to include imports and aspects like function signatures in its consideration.
3) **Refactoring** Tasks of this kind require more than the creation of new solutions: they might involve the restructuring existing components. Hence, the subject have to consider the existing (old) behavior and has to perform a refactoring in an at least limited sense.

We plan to use this classification later for a more differentiated result analysis. In the following, we give more detailed description of the components for a better understanding of the project and experiment.

- **Camera** This component uses the camera sensor of the Simbad framework to detect whether a red ball is located in front of the robot. It does so by analyzing the color picture delivered by the hardware. The result of the analysis is reported through its output port `ball`. The image perceived by the camera is provided in form of a 100×100 array of RGB values. For image recognition, a simple algorithm is supposed to be implemented by the subjects. Instead of using an actual object recognition algorithm, only the red channel of the image is used. Furthermore, only the center column of pixels is considered and segmented into sections with size of 10 pixels. Since the virtual environment is mainly colored in green and blue, an average red value in one of these sections larger than the threshold 100 is sufficient to confirm a ball in from of the robot. This way, a satisfying trade-off between task complexity and detection accuracy is achieved.
- **Collision** This sensor is used to detect physical contact of the robot to other objects in the environment. For this purpose, it is equipped with a belt of 16 evenly distributed bumper sensors. To signalize whether a collision was detected the `collision` port is used. Each sensor reports a collision through Boolean value and the component iterates over all sensors to determine whether any sensors reports a collision.
- **SearchControl** This is a composite component consisting of the subcomponents `InternalControl` and `Logger`. It is the central entity of the project since and is in charge of controlling the robot. To facilitate the removing errors, its behavior as well as the received input values are logged through `Logger`. This component does receive any treatment from the subjects.
- **Logger** This component handles the logging requirement of the `SearchControl` type. Therefore, it receives all input values from the containing component as well as the output values of the `InternalSearchControl`. Monitoring their changes, it writes these events including a time stamp to a log file.
- **MovementControl** This composed component controls the robot's movement. Therefore, it translates the movement commands received from `searchControl` to movement of the actual "physical" devices of locomotion. At the beginning of the experiment, it contains instances of `MotorControl` and `CompositeMotor`, the latter matches Simbad's default kinematic model. During the experiment, the participants have to restructure `MovementControl` and replace the `CompositeMotor` instance with two instances of type `WheelMotor`.
- **MotorControl** The task of this component is the translation of movement commands for the actual drive control. Initially, when the default kinematic model is deployed, the component possesses two outgoing ports to control an instance of `CompositeMotor`. After restructuring, the differential model is used, it features one output value for each of the two `WheelMotor` instances.
- **CompositeMotor** This component corresponds to one of the kinematic models available in Simbad and represents a single "physical" propulsion. It receives a `velocity` and a `rotation` value as input to control the movement. The behavior implementation of this component passes the two incoming values to the Simbad framework.
- **WheelMotor** A component of this type matches with one wheel of the differential kinematic model. It receives one value as input. A positive one results in a forward rotation of the wheel, while a negative value causes a backwards one, respectively. Consequently, `CompositeMotor` poses implementation challenge regarding integration of native API.

F. Conduction Plan

The complete experiment takes 100 minutes per participant. For the execution, experiment equipment is provided, this includes notebook computers running Ubuntu Linux, the Atom[2] editor with disabled syntax highlighting and disabled language completion support, and a makefile responsible for triggering the maven-driven build processes. We provide the restricted editor and the makefiles to mitigate the effects of advanced IDEs. Using basic editing functionalities, such as "find" or "replace" is not restricted. The provided makefiles take care of parsing models, generating code, compiling, and packaging it.

To enable a correct attribution of the subjects actions to the correct metric, we use screen recording and employ a camera to capture the non-digital behavior. Furthermore, we are storing the project state at each compilation and run for an easier analysis in the aftermath.

During execution, each subject performs the following:

[2]Atom website: https://atom.io

1) **Pre-questionnaire (5min)** Before the actual start of the experiment, the participant is asked to fill out a survey on his or her modeling and programming skills. This includes inquiring the experience with modeling and object-oriented programming in general as well as specific modeling and programming techniques and languages.
2) **Introduction (25min)** Next, we give an introduction to the study's development environment, MontiArc-Automaton, and the respective behavior modeling techniques for the participants group. Moreover, we introduce the `BallFinder` and the modeling task. The participant may explore prepared example code to familiarize himself or herself with the assigned modeling technique and may inspect the provided architecture of the experimental project.
3) **Tasks (65min)** We hand out the task description to the participant and let him or her perform the task using the assigned modeling technique. The participants are requested to solve the task steps in order of their presentation. This order is identical to the presentation in Table I, the subjects will therefore implement the robot following the flow of information from input to output. During the whole time, the participants are allowed to ask questions related to the understanding of the exercises, but no information regarding the solution is given.
4) **Post-questionnaire (5min)** Ultimately, we conduct a second questionnaire to learn more about the acceptance and ask for general feedback about the experiment.

IV. Discussion

Our study design enables comparing MDE with integration of handcrafted artifacts to pervasive MDE with action languages. As representatives, we chose AJava and the delegation pattern for the conduction of the experiment. This study should give us a first good indication about their respective benefits in practice. To this effect, we embedded the experiment's tasks into a small project to simulate a realistic scenario. Through a minimal set of tools, we ensured that we compare the techniques and not the tooling. By our choice of tasks, we covered many different aspects of the development process. In addition to the implementation of logic, we should receive insights about refactoring and the integration of native API as well. For the comparison, we do not only consider performance metrics, we also take acceptance and usability into account. Both aspects add up to full picture, compensating shortcomings of other metrics. Otherwise, a later employment of the technique in practice could be endangered.

The greatest risk to an experiment is the invalidation of results found in the collected data. Therefore, we discuss the possible threats to internal and external validity in the following and examine how we assured that conclusions are legit and in which context the results are valid.

A. Threats to Internal Validity (Causality)

Threats to internal validity arise from factors influencing the causality between independent and dependent variables [21]. We recognize following potential threats:

- **Knowledge of Java** Especially the performance metrics to answer **RQ1** and **RQ2** would be affected. Since we expect a mostly homogeneous group of participants with this regard and we randomize the participants among the groups, we do not consider this as problematic. Questionnaires are conducted to support this assumption.
- **Experience with delegation pattern or AJava** Similar to the Java knowledge, we expect no risk from this aspect. Differences in experience should not be present in large, for this claim we use the questionnaires as well and provide a proper introduction to the techniques before the actual experiment.
- **Tooling support** Tools, such as IDEs, facilitate the development and, hence, entail the risk of actually measuring the quality of the tools instead of the techniques to be investigated in the experiment. Through minimizing the tooling and providing the identical tooling to both groups, we ensure that this does not affect our experiments.
- **Unfamiliar environment** As a side effect, we force the subjects into using an unfamiliar development environment, which may be conceived a hindrance. To address this, included an introduction phase in which each subject can make herself or himself comfortable with the environment. As this applies to both groups, however, this should not impact the results.
- **Absolute acceptance** The performed acceptance measurements are absolute and therefore not usable for a comparison unless the same subject experiences both techniques. We do not negate this finding. However, we would like to note that we are not interested in determining an absolute acceptance value and if one techniques performs really bad or is dissatisfying in certain tasks, we still should be able to notice a difference.
- **Small sample size** The number of participants makes the results unreliable in terms of significance. There is not doubt about this risk, but we would like to note that the exact indicators are only available after the execution of the experiment. Furthermore, we employ multiple tasks for this reason and perform the comparison for each individually, but also in sum. When increasing the sample by including more external participants, we see the risk of create more heterogeneous groups endangering our controlled variables.

B. Threats to External Validity (Generalizability)

Threats to external validity address the generalizability of empirical research. For our study, the most important threats to its generalizability are:

- **Project size** The software engineering challenges of our project is several magnitudes less than in industrial projects. Hence, the results can not be generalized easily. The size is unfortunately naturally limited by the available time of the participants in a laboratory experiment. We tried to overcome this problem by embedding different

types of implementation challenge (see Table I) to cover multiple application scenarios.

- **Academic context** This experiment features an artificially created project which differs from real-world projects. Again, we refer to the different implementation challenges we employed. Beyond, only repetitions in industry are able to tackle this threat.
- **Non-professionals** We incorporated students in our experiment which might not be a good replacement for actual professionals. To address this issue, we refer to [22] for reasons why we think that choice restricts the generalization only minimally.
- **Choice of techniques** We used Java, AJava and the delegation pattern to compare pervasive MDE to integrating handcrafted code. One might argue that the delegation pattern is especially complicated and using other patterns would yield better results for the approach of handcrafted code. Or that Java might be especially unsuited for this topic in general. As Java is one of the most popular languages for professional software engineering and the delegation pattern is very popular for integrating generated and handcrafted code, we assume that both reflect the state of the art appropriately.

V. Preliminary Results

We conducted a pilot study with 13 participants, out of which 7 employed the delegation pattern and 6 employed AJava to developed the `BallFinder` robot software. Interesting findings are illustrated in Figure 3 and discussed below.

Regarding **solution finding time**, the AJava group solved their tasks in a median of 39 minutes compared to 64 minutes required by the delegation pattern group. With respect to **programming time**, the AJava group required a median of 30 minutes compared to 55 minutes of the delegation pattern group. With p-values of less than 1%, both **H1.1**$_0$ and **H1.2**$_0$ hence can be rejected for the pilot study. For the variables of **error resolving time** and **testing time**, we received indifferent results without relevant significance, hence we cannot refute the related hypotheses. The same holds for the number of **code generation iterations**: On average, the AJava group generated code 15 times, whereas the delegation pattern group on average generated code 20 times.

Developing with AJava led to a consistent number of between 11 and 14 **test errors**, whereas participants of the delegation pattern group finished with a wide range of between 1 and 14 errors. Measuring the amount of **test errors per minute** produced similar diversity, hence **H2.2**$_0$ and **H2.3**$_0$ cannot be rejected either.

The participants' assessment on the different techniques' **convenience** generally seem favor using AJava: 83% of the participants of this group consider AJava practical, whereas only 43% of the delegation pattern group participants consider that practical. As the results are almost significant only, we also cannot refute **H3.1**$_0$. Considering the **overhead** imposed by the respective techniques, 50% of the AJava group's participants consider AJava not introducing overhead at all, whereas

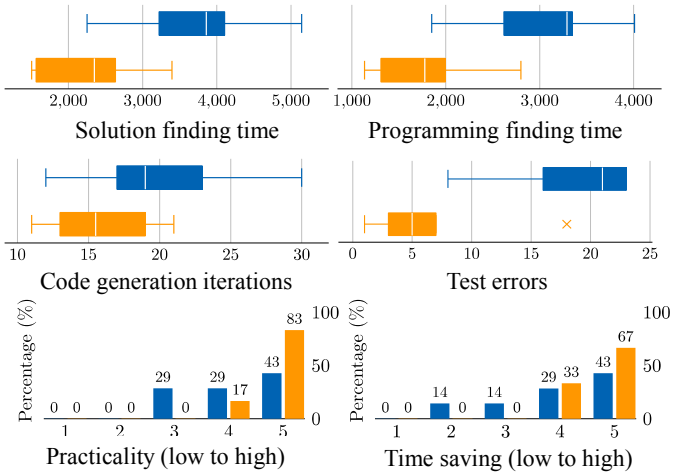

Figure 3. Findings from performing our study with a pilot group of 13 students.

57% of the delegation pattern group consider it introducing large overhead. Significance of the results implies that **H3.2**$_1$ can be accepted. Regarding the findings of **time saving** with any of the techniques and **demand for tooling**, we did not receive significant results or notable biases in the results. Hence we cannot refute the related hypotheses.

VI. Related Studies

Our study design basically compares two "programming" languages. From this vantage point, several related studies have been conducted. The study presented in [15], for instance, compares "tangible" graphical programming languages where the resulting programs were executed on a real robot platform. The authors do not investigate the differences in abstraction between different languages.

The authors of [16] compare seven GPLs regarding runtime efficiency, loading efficiency, program length, and other factors. They also do not compare their levels of abstraction.

Other studies investigate the introduction of MDE in industrial contexts to identify obstacles [17] or challenges [18] for its adoption [19]. They do, however, not directly investigate the acceptance of pervasive MDE over the programming techniques established at the respective companies.

There also are various studies on different modes of (virtual) robot programming, such as the experiment reported in [20], which are related to our experiment, but not to the investigated objects.

Overall, to the best of our knowledge, there is only little related work on comparing pervasive MDE to providing system behavior via integration of handcrafted GPL code.

VII. Conclusion

We have presented the design of a study on the acceptance and effects of pervasive model-driven engineering using action languages to interface GPL code artifacts. In the proposed study, two groups of participants will develop the behavior of a robot fetching balls in a simulation environment using

MontiArcAutomaton with either embedded Java/P or by integrating handcrafted artifacts. We record the study sessions and instrument these with quality assurance infrastructure, and surveys. Based on this data, we aim to gain a better understanding about the impact of using action languages in modeling activities. Preliminary results tend to favor pervasive MDE with respect to efficiency and convenience. However, these results are based on a small sample size and must be investigated with larger groups of participants.

REFERENCES

[1] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," *Future of Software Engineering (FOSE '07)*, no. 2, pp. 37–54, may 2007.

[2] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, K. Czarnecki, and B. von Stockfleth, *Model-Driven Software Development: Technology, Engineering, Management*, ser. Wiley Software Patterns Series. Wiley, 2013.

[3] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. Mir Seyed Nazari, K. Müller, A. Navarro Perez, D. Plotnikov, D. Reiß, A. Roth, B. Rumpe, M. Schindler, and A. Wortmann, "Integration of Handwritten and Generated Object-Oriented Code," in *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, ser. CCIS, vol. 580. Springer, 2015, pp. 112–132.

[4] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[5] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, "Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems," *Journal of Software Engineering for Robotics*, 2015.

[6] B. Rumpe, *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[7] H. Krahn, B. Rumpe, and S. Völkel, "MontiCore: a Framework for Compositional Development of Domain Specific Languages," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, no. 5, pp. 353–372, September 2010.

[8] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation." in *International Conference on Software Engineering (ICSE) 2002*. ACM Press, 2002.

[9] A. Haber, M. Look, A. Navarro Perez, P. Mir Seyed Nazari, B. Rumpe, S. Völkel, and A. Wortmann, "Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components," in *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, 2015.

[10] J. O. Ringert, B. Rumpe, C. Schulze, and A. Wortmann, "Teaching agile model-driven engineering for cyber-physical systems," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ser. ICSE-SEET '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 127–136.

[11] R. Heim, P. M. S. Nazari, J. O. Ringert, B. Rumpe, and A. Wortmann, "Modeling Robot and World Interfaces for Reusable Tasks," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 1793–1798.

[12] M. Fowler, *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[13] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 728–738, Nov 1984.

[14] L. Hugues and N. Bredeche, "Simbad: An Autonomous Robot Simulation Package for Education and Research," in *Proceedings of the 9th International Conference on Simulation of Adaptive Behavior (SAB)*, ser. Lecture Notes in Computer Science, vol. 4095. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 831–842.

[15] M. S. Horn, E. T. Solovey, R. J. Crouser, and R. J. Jacob, "Comparing the use of tangible and graphical programming languages for informal science education," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 975–984.

[16] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, Oct 2000.

[17] M. Staron, "Adopting model driven software development in industry-a case study at two companies," in *MoDELS*, vol. 6. Springer, 2006, pp. 57–72.

[18] P. Baker, S. Loh, and F. Weil, "Model-driven engineering in a large industrial context – motorola case study," *Model Driven Engineering Languages and Systems*, pp. 476–491, 2005.

[19] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 633–642.

[20] C. S. Tzafestas, N. Palaiologou, and M. Alifragis, "Virtual and remote robotic laboratory: Comparative experimental evaluation," *IEEE Transactions on education*, vol. 49, no. 3, pp. 360–369, 2006.

[21] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[22] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "Empirical software engineering experts on the use of students and professionals in experiments," *Empirical Software Engineering*, 2017.