A Controlled Experiment for Evaluating the Comprehensibility of UML Action Languages

Omar Badreddin
Department of Computer Science
Northern Arizona University
Omar.Badreddin@nau.edu

Maged Elaasar
Department of Sys. & Comp. Eng.
Carleton University
melaasar@gmail.com

Abdelwahab Hamou-Lhadj
SBA Research Lab
ECE, Concordia University
abdelw@ece.concordia.ca

Abstract—**Action Languages represent an emerging paradigm where modeling abstractions are embedded in code to bridge the gap with visual models, such as UML models. The paradigm is gaining momentum, evident by the growing number of tools and standards that support this paradigm. In this paper, we report on a controlled experiment to assess the comprehensibility of those languages and compare it to that of object-oriented (OO) programming languages. We further report on the impact of also having access to the UML notation on the comprehensibility of those languages. Results suggest that action languages are significantly more comprehensible than traditional OO languages. Furthermore, there was not a significant improvement in comprehensibility when the UML notation was used along with both OO and action language code. We conclude that action languages are a promising alternative to traditional OO languages for specifying details, yet seem to be as comprehensible as high-level visual models.**

***Keywords: UML, Model Driven Development, Alf, Object Orientation, Model Oriented Programming Languages*.**

I. INTRODUCTION

The UML lacks formal execution semantics for many of its elements [28]. For example, UML use case modeling notation does not map directly to any executable semantics. Careful investigation of many other modeling notations reveals similar execution semantic gaps [29]. A UML action language gives unambiguous execution semantics to a subset of UML. An example of such language is Alf, a textual action language for Foundational UML (fUML) [1].

Action languages and UML share some commonalities. Both of them are an attempt to deal with the ever-increasing complexities of system development through abstraction. UML provides a visual notation that abstracts away the structure and behaviour of the system. It also promises some level of portability, as UML models can typically be used to generate source code for multiple platforms.

Action languages, such as Alf, are designed to be high-level executable languages. Like UML, they allow the definition of the key abstractions of the system, but they also provide mechanisms to specify the system's detailed behaviour similar to traditional OO languages. For example, in an action language, the developer can declaratively define the concepts of a system with classes, their inter-relationships with associations, and their behaviour with state machines. The detailed activities performed in each state can be specified imperatively with executable code.

Action languages engage users in a familiar textual and executable environment (without the need for forward or reverse engineering processes between model and code). They bare many similarities with modern OO languages like Java and C++, which provides significant value for rapid system prototyping. While the comprehensibility of the UML notation has been well investigated before (see [4]), to our knowledge, there is no study that investigates the comprehensibility of UML action languages compared to OO languages. More particularly, we are interested in investigating the following research question:

RQ1: How do the emerging UML action languages compare to traditional OO languages in terms of comprehensibility?

This question investigates whether or not there is a significant difference in the way software engineers understand action languages compared to OO languages. In addition, we are interested in investigating if there is added value in combining the UML visual notation along with action or OO languages. For this, we ask the following research question:

RQ2: What is the incremental impact on comprehension when combining the visual UML notation with action language or OO languages?

To answer these questions, we designed a controlled experiment where participants were given samples of code expressed in action languages and OO languages that were extracted from an open source software project [17]. The participants were asked to complete a set of tasks, ranging from answering simple comprehension questions, to performing debugging activities. The experiment used two action languages and two OO languages. Also, relevant models in UML notation were also made available to assess the added value on comprehension.

Our findings show that action language code is more comprehensible when compared with OO code. Furthermore, the experiment did not show any significant increase in the comprehension of either OO or action code when coupled with UML models.

The remaining parts of the paper are organized as follows. In Section II, we provide background on the two action languages that are used in this experiment. We present, in section III, the experiment's setup and design based on the guidelines for reporting experiments in software engineering proposed in [31]. In Section IV, we present the results and analyze them quantitatively and qualitatively. We discuss threats to validity in Section V. In section VI, we review related work. Finally, we conclude and outline future work in Section VII.

II. Background on Action Languages

Action languages are typically textual and support abstractions such as classes, associations, multiplicities, and state machines. We believe there are two main motivations behind the emergence of action languages. First, action languages help bridge the gap between less abstract object-oriented languages, and more abstract modeling notations. For example, in a UML model, one can define classes, their relationships (e.g., with associations) and their behaviour (e.g., with state machines). However, in a typical object-oriented programming language, such as Java or C++, one is unable to directly manipulate those abstractions. For example, one cannot express associations between classes or the exact multiplicities of collection properties. Also, while it is possible to specify state machines as the behaviour of classes in UML, one cannot express the same level of abstraction in the corresponding OO code. (Note that the mapping of such modeling abstractions to object-oriented languages varies from one approach to the other.) The developer has to learn how such abstractions are mapped to a programming language to be able to manipulate them in the OO code. This leads to a wide gap between programming and modeling languages.

The second motivation for action languages is a growing realization of the software developers' preference to use familiar textual environments [6][26]. Code, unlike models, has a serial nature and might be easier to maintain with any text editors. Developers do not need to worry about layout, as is the case with visual notation. In addition, wide adoption of code repositories (e.g., Git) means that code remains the main development artifact [6].

One can argue that the first trace of the emergence of textual modeling language is Human-Usable Textual Notation (HUTN) [8]. This effort was sponsored by OMG (Object Management Group), but later lost momentum and has been abandoned. More recently, in 2008, OMG issued a Request For Proposal (RFP) for a concrete syntax for a UML action language, which was referred to, at that time, as UAL [9]. The RFP requirements included support for the Foundation subset of UML (fUML). Two proposals were submitted, one from IBM and one from Mentor Graphics. The two proposals were later combined and named Alf, Action Language for Foundational UML [10].

In parallel, multiple industry and academic efforts were investigating textual modeling, textual representations for UML, and action languages. For example, TextUML [11] provides an equivalent modeling capability where models are represented textually. SOIL [7] is a language that allows the embedding of OCL-like statements into programming languages. Another notable effort is Umple [17], a language that embeds UML modeling abstractions textually in object-oriented code.

In this work, we selected Alf and Umple as two instances of action languages. Alf was selected because it is sponsored by OMG, which has in October 2013 published an updated standard for the language [1], with promising tool support [32]. Umple was selected because it provides tooling for an executable action language environment that is open sourced. Both choices enabled us to setup our experiment's environment with necessary tools.

The following two subsections provide a brief background on both Alf and Umple. The background is only sufficient for the purpose of introducing the experiment. The reader is encouraged to refer to other publications on Alf [12] and Umple [13] for more information.

To show how the two action languages work, we reuse a subset of the example used in the latest Alf published standard on page 379 [14], which itself is borrowed from a book named Executable UML: A Foundation for Model Driven Architecture [15]. Our example model consists of two classes, Order and Customer (Figure 1). A customer may have one or more orders, and an order may or may not be associated with a customer.

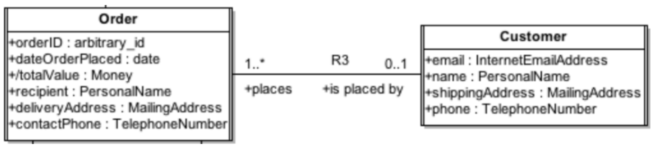

Figure 1 Example in UML Notation [14]

A. *Alf Action Language*

Alf represents the Order class as in Figure 2 below:

```
active class Order {
  public orderID: arbitrary_id;
  public dateOrderPlaced: date;
  public totalValue: Money;
  public recipient: PersonalName;
  public deliveryAddress: MailingAddress;
  public contactPhone: TelephoneNumber;
..
```

Figure 2 Example Alf code

The representation is very similar to Java and C++. This is an intentional design objective of Alf and is meant to enhance adoption by software developers who are already familiar with OO languages. What is new in Alf is that it supports the representation and manipulation of modeling abstractions. The Alf code snippet in Figure 3 shows how the association between Order and Customer is represented:

```
public assoc R3 {
  public places: Order[1..*];
  public 'is placed by': Customer[0..1]; }
```

Figure 3 Example Alf association

Typical object-oriented languages do not support such explicit representation of associations. Alf, in addition, provides syntax for manipulating state machines. The class *Order* is an active class, meaning that its behaviour is specified by a state machine, which Alf also defines as part of its textual syntax. The state machine is defined on page 380 of the Alf published standard [14]. The Alf standard includes mechanism to specify imperative statements in various places including the states' entry/exit/doActivity actions. Such statements are similar to those expressible with high-level programming languages like Java and C++.

B. Umple Action Language

Umple's syntax is similar to Alf and very similar to object-oriented languages. The difference between Alf and Umple is in the syntactic representation of modeling abstractions and in the approach of bridging the gap between them and the code. Figure 4 is Umple's representation of the same class diagram in Figure 1.

```
class Order {
  1..* -- 0..1 Customer;
  int orderID;
  date dateOrderPlaced;
  recipient;
  address deliveryAddress;
  .. }
```

Figure 4 Example Umple code

Notice that Umple, unlike Alf, allows the definition of the association between *Order* and *Customer* to be in either the *Order* class (Figure 4) or the *Customer* class. Both Alf and Umple, however, allow the definition of the association to be separate of either class. Also, property types in Umple can be implicit. For example, property *recipient* has an implicit default type of String in Figure 4. Also unlike Alf, Umple does not provide its own expression syntax but uses that of modern high level programming languages as-is.

The representation of state machines in the two languages is different. Alf provides syntax for specifying state machines and their various expressions (e.g., the transitions' guards and the entry/exist/do behaviours) declaratively. Umple, on the other hand, provides syntax for defining state machines but relies on the embedding OO language syntax to specify the various expressions. This difference is significant, and this is the motivation for utilizing two languages to represent action languages. Nevertheless, the specifics of the distinction between these two languages is not of concern with respect to this experiment.

III. EXPERIMENT DESIGN

The goal of this experiment is to evaluate the comprehensibility of action languages in comparison to traditional OO programming, and to evaluate the added comprehension value of typical visual notations such UML. An important presumption here is that an action or an OO language does not replace the need or role of a visual UML notation for key system components, relationships, and behaviour. Therefore, part of this experiment is designed to evaluate the comprehension added value of the UML visual notation.

A. Experiment Artifacts

In this experiment, we use two systems specified in two action languages (Alf and Umple), two OO languages (Java, and C++), as well as UML. This means we have 10 artifacts in total. We discussed the rationale for using Alf and Umple in the previous section. We selected Java and C++ because of their popularity and wide use in practice. Also, these languages do not differ significantly in syntax or abstraction level, which helps keep our experiment design balanced. UML is used as a reference notation for visual modeling.

The two systems used in this experiment are extracted from the Umple's open source project [17]. The first one is a subset of the UML class diagram metamodel and the second one is a subset of the UML state machine metamodel. These two systems are selected because they provide a suitable mix of modeling abstractions (e.g., classes, properties, etc.) and their implementations (e.g., constructors, getters, setters, etc.). The size of these systems is suitable for the purpose of the experiment as well. The systems can also be effectively represented using the different notations being evaluated in this experiment, i.e., Alf, Umple, Java, C++, and UML.

We have opted to use a subset of the UML class and state machine metamodels to keep the experiments simple. Also, we focused on the abstract syntax metamodels and not the visual (concrete syntax) specifications.

The experiment artifacts were first examined by three independent researchers who are not involved in this study. The researchers checked the experiment artifacts for consistency, i.e., made sure that the artifacts are semantically equivalent. They also checked the coding and modeling styles to ensure that typical ones are used. The reviewers sent their recommendations to us. We then evaluated them and updated the artifacts as necessary. This process was iterative until all three researchers agreed that the models and code are consistent and representable.

Table 1 summarizes the key properties of the experiment artifacts. The table lists the number of lines of code for Java, C++, Umple and Alf. For UML, the number of modeling elements is listed.

TABLE 1. NUMBER OF LINES FOR EXPERIMENT ARTIFACTS, AND NUMBER OF MODEL ELEMENTS FOR UML

System	**Java**	**C++**	**Umple**	**Alf**	**UML**
Class diagram metamodel	196	192	151	157	129
State machine metamodel	172	180	140	142	117

The artifacts were presented to the participants as follows: Both Java and C++ code snippets were presented in an Eclipse IDE, showing the typical code canvas, outline view, and problems view. Both Umple and Alf were presented in a custom-designed eclipse environment. The environment was developed to match those of Java and C++. Alf and Umple's environments contained code canvas with typical highlighting of code, outline view, as well as problem view. However, advanced editing and code-assist features that are available to Java and C++ were not available to Alf and Umple. We are not concerned about such limitations in the case of Alf and Umple. First, the experiment duration is relatively short, and our observations indicate that participants do not get to use advanced editing features in any significant manner. In addition, such limitations do not affect our hypotheses, since any bias will only make our conclusions stronger.

UML visual models were presented as images only (i.e., not in a UML tool). The images were approximately the same size as the code canvases used. Two UML models were used that represent subsets of the class and the state machine metamodels. We arrived at those subsets by removing what we judged to be ambiguous or less familiar, elements. For example, we removed elements that represent protocol state machines. In addition to the modeling abstractions selected, the experiment artifacts included implementation code. For brevity, the UML models and implementation code are not shown here. However, such models and Java-version of the implementation code are published as part of Umple's open source project [17].

Java and C++ implementations of both metamodels were developed by us (the Java one was in the context of the Umple project), and reviewed by three independent researchers to ensure consistency and reasonable implementation choices. Each reviewer was asked to report inconsistencies and implementation concerns to us. We either implemented the change, or revaluated the comments of the reviewer by involving a third reviewer. Eventually, all three reviewers agreed that the two representations were consistent and were semantically equivalent to the corresponding subsets of the UML metamodels.

The Alf implementation of the two systems was more challenging for two reasons. Alf is an emerging standard where the syntax is being continuously revised. We adopted the syntax published in the OMG standards as of October, 2013 [14] and stuck to it, even though we are aware of other variations and proposals that are underway. The second reason is because Alf is not widely adopted yet, and it was not possible to look at existing code to find out whether there is a consensus on common coding patterns. We selected what we found to be the most natural syntax alternative and used it for both systems.

The effort to build Umple representations of the two systems was relatively simpler for us. Umple's code is published as an open source project [17] and contains an implementation for class and state machine metamodels.

B. *Participants*

The experiment involved 32 participants that we divided into two groups. All participants received the same artifacts. The only difference was whether the participants had the visual notation of the system in UML or not. This way, we can evaluate the effectiveness of the action languages as compared to the OO languages, as well as assess the added value of having UML notations in combination with the system code (C++, Java, Umple and Alf).

We should note that we did not consider assessing the comprehensibility of UML artifacts alone. This decision was motivated by the following: UML is not meant to replace the need for code, whether this code is OO or Action Language; UML is typically used in conjunction with code, which is the paradigm used by most UML modeling tools, such as IBM's Rational Software Architect and Papyrus. Instead, we were interested in answering the question (RQ2) of whether UML notation adds to the comprehensibility of textual languages. We discuss this in the results section of this paper.

The participants were software engineering or computer science students as well as software engineering practitioners. In total, 32 participants were recruited, out of which 14 had a PhD degree in a related field, two had a Master degree, and the rest had a Bachelor degree. We collected their experience and background levels on a scale from 1 (beginner level) to 5 (expert level). Their average knowledge of Java was the highest (3.3/5.0), followed by C++ (3.1/5.0), followed by UML (2.7/5.0), followed by Umple and Alf (1.7/5.0).

We analyzed the data using different participant slices. One slice was based on education levels: those with a PhD only, those with a Master degree only, and those with a Bachelor degree only. Another slice is based on the level of knowledge of the languages under study. We found the results of analyzing the data for these slices not significantly different than the results for the entire population.

Participants were recruited randomly using convenient sampling techniques. Recruitment was announced on multiple news boards. Appointments were scheduled based on participants' availability. Selection criteria included having a degree in software engineering or a related field, having familiarity with UML and action languages, and having worked as a professional software engineer for at least one year. Participation was both anonymous and voluntary. The identity of participants was never collected. Throughout the study, we reminded the participants that they can stop participation at any step. Participants were not compensated for their participation. The experiment is conducted after proper approvals had been obtained.

C. *Questions and Task Lists*

We designed a total of eight questions and four tasks that range from simple comprehension questions, to performing tracing and debugging tasks. The questions were uniform across the different artifacts. However, there were only minor variations in wording of the questions and tasks between those posed for C++ and Java and those posed for Alf and Umple. The variations were minimal and we do not expect such variations to affect the results of the experiment. In fact, during the pilot study, our reviewers made comments that made us do such minor wording changes.

The questions and tasks for the first system were significantly different than the questions and tasks for the second system. This is simply because the two systems are significantly different. This difference is by design and is intentional. However, we maintained some level of relevance in the two sets. We made sure that the number of questions and tasks and their relative complexity are similar. This enabled us to analyze the results for both systems consistently. Table 3 shows an excerpt of the set of questions and tasks used for the state machine system.

TABLE 2.EXCEPRT OF QUESTIONS FOR THE STATE MACHINE METAMODEL

	Question / Task
Q1	How many activities can a state have?
Q2	How many transitions can be associated with a state?
Q3	Can you create a transition from one state to multiple states?
T1	Create a state machine to represent the UML model in figure 1.

T2	Create a guard condition to resolve the ambiguity in this model. Note you may first need to identify the ambiguity in the model.
T3	Is this model complete or incomplete? If it is incomplete, suggest a way to complete the model and implement the change.

Each participant attempted the questions and tasks of the two systems (see next subsection). The first system was the class diagram metamodel, whereas the second system was the state machine metamodel. We believe that the learning effect of the first system had minimal impact on the second system due to the different nature of the systems (class diagram is for structural modeling vs. state machine diagram is for behavioural modeling). Not all participants were assigned all artifacts. Also, the assignment of artifacts to participants was not left up to the participants. Rather, it was controlled by us with the intention to make the experiment design balanced.

Participants were not given the question lists in advance to minimize the risk they may look at other questions while attempting to answer the current question. Participants were given the choice between a Windows laptop and a Mac laptop. Their preference was always accommodated. This is because we wanted to make sure that a familiar environment is provided for each participant. However, participants were not allowed to use their own laptops. This was due to the effort required to set up the environment, the experimental artifacts and the recording software. The questioning sessions were audio recorded. Time was measured starting from the end of posing a question until the participant finished answering the question. We also recorded the laptop screen in video from the beginning of the experiment and until the end.

At the onset of the experiment, participants were asked a number of profiling questions about their background, prior knowledge of C++, Java, Alf, Umple and UML. We also collected information on their software engineering courses and work experience. The objective of this profiling information is to analyze it along with the experimental data and examine any bias caused by the experiences of the participants. We disqualified participants who did not meet the minimum participation requirements.

We should mention that at the beginning of the experiment, participants were shown three short videos introducing UML, Alf, and Umple concepts. However, we did not expect it would influence the experiment results much in favour of those languages.

D. Study Design

In this section, we state the research questions, variables, and analysis methods used in this experiment.

RQ1: How do the emerging UML action languages (Umple and Alf) compare to traditional object-oriented languages (Java and C++) in terms of how easy to understand and use?

We state the following hypothesis:

H1: A system specified in Umple or Alf is more comprehensible than an equivalent specification of the system in Java or C++.

In other words, participants take on average less time to answer questions when presented with a version of a system implemented in an action language as opposed to a Java or C++.

The corresponding null hypothesis is:

H1o: Action languages and object orientation do not differ in comprehensibility.

H1 is a baseline. If we can reject the null hypothesis then we can be confident that there is a difference in comprehensibility.

Variables: The independent variables are the notation of the two systems used in this study with values: 'C++', 'Java', 'Alf', and 'Umple'. The focus was on measuring the comprehensibility of the languages. Comprehension was measured by eight questions and four bug fixing tasks. The dependant variables used to measure comprehension are:

- **Time**: The time taken to respond to a question or provide a fix for the task, measured in seconds.

For the fixes, the participants continued to edit the code until the correct answer is reached. This is either when the participant recognizes that he or she had accomplished the task, or when we recognized that the bug is fixed and notified the participant.

- **Quality**: The quality of the answer or the fix, which is a subjective measure. This is collected for meta-analysis, and is assessed by two independent reviewers. If the evaluation of the two reviewers does not match, a third reviewer is involved to make a decision based on his or her judgement, as well as the evaluation of the two previous reviewers.

Analysis:

We use descriptive statistics to compare the time it takes to answer the questions or perform the fixes using C++/Java to the time it takes to do the same in Umple/Alf. We also use a two-tailed t-test to measure the *statistical significance* between the average times it takes using both paradigms. As confirmatory evidence (in case of significant departure from the normality requirements of the t-test), we apply the Mann-Whitney test (U-test).

RQ2: What is the added value of the visual UML notation when used with action language or OO languages?

We state the following hypothesis:

H2: UML visual notation enhances comprehension when used with action or object-oriented code.

The corresponding null hypothesis is:

H2o: UML notation does not enhance comprehension when used with action or object-oriented code.

Variables: Similar to the previous questions, we use independent variables, which are the notations of the two systems used in this study with values: 'C++', 'Java', 'Alf', and 'Umple'. We measure comprehension the same way as before. The only difference is that this time, we provide the UML notation with the artifacts. We compare the answers provided by participants that used UML notation with those of the participants that did not use UML notation **(**RQ1).

Analysis:

We used descriptive statistics to compare the time it takes to answer the questions and perform the tasks using C++/Java/Umple/Alf with UML notation to the time it takes to do that without UML notation. Similar to the previous question, we also used a two-tailed t-test to measure the *statistical significance* between the average times with or without UML. The Mann-Whitney test (U-test) was used in case of significant departure from the normality requirements of the t-test.

E. *Design Validation – Pilot Study*

In order to initially verify and validate the design of the experiment as well as identify potential flaws in the design, we conducted a pilot study. The pilot study was conducted using eight other participants, who were selected based on availability and software engineering background. The pilot data was excluded from the analysis.

This pilot study was very instrumental in refining many aspects of the experiment. For example, we found that some of the original wording of the questions was not clear. It was also found that participants tend to become less active by the end of the experiment. The question wording was corrected and reviewed independently again. The reduced activity was mitigated by reducing the number of questions and giving participants a break between the two systems.

IV. Results and Analysis

In the course of the experiment, each participant was given two rounds of questions and tasks corresponding to the two systems. Each participant spent on average 70 minutes. The shortest duration was 49 and the longest was 83 minutes, this included a 5-minute break between each system (round) and the time the participants took to read and sign the consent documents.

Participants were given a laptop that guided them through the experiment. An HTML application was developed so that participants can click next when they are finished with their answer. Video and audio recording software was running in the background. The audio is recorded to provide hints in case of exceptional situations occurring. For example, the audio was used in case the experiment operations were interrupted by the request of the participant. The video recorded the screen, and was used to measure the time durations for each question. The distribution of artifacts was balanced, so that equal number of participants answered questions on equal number of notations.

The overall average for answering the questions and performing the bug fixing tasks was 47.1 seconds. This is in line with our pilot study, and is in line with our design objectives, which is keeping the questions and tasks relatively simple so they each can be answered within 3-minute on average. The standard deviation was 15.6 seconds. Fig. 5 summarizes the experiment results.

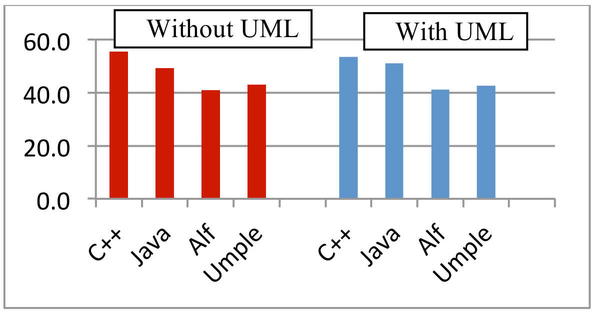

Fig. 5. Overall experiment results

From Figure 2, a few patterns immediately become evident. First, the average results for both the 'With UML' and 'Without UML' cases are almost identical for both the action languages and the OO languages. This suggests that having UML notation does not improve comprehension, which is an unexpected result. We discuss our interpretation of this result in the discussion section of this paper.

Also evident from this quantitative analysis is that Java seems to have slightly outperformed C++ (likely due to the experience of the participants). Also both Alf and Umple have performed better than the OO languages. This seems to suggest that being at the model level provides comprehension and usability benefits to action languages.

Furthermore, (as shown in Table 4), the standard deviation (SD) for the OO languages (16.6 seconds) was higher than the SD for the action languages (12.1 seconds). This we believe is due to participants having different levels of experience with those OO languages, while almost similar experience with actions languages. Also, the SD for both systems, 'With UML' and 'Without UML', is 15.6 seconds. This implies that differences between the two systems were not significant, which is counter-intuitive. We discuss our interpretation of this result in the discussion section of this paper.

TABLE 3. RESULTS SUMMARY

	Without UML				With UML			
	C++	Java	Alf	Umple	C++	Java	Alf	Umple
Average	55.4	49.3	40.9	42.9	53.6	51.5	41.2	42.7
SD	18.1	16.8	16.1	11.6	18.8	16.3	15.7	11.7
Overall Average	47.1				47.1			
Overall SD	15.6				15.6			

Table 5 shows the time averages of answering the questions. One objective of the design of this experiment is to keep the questions and tasks of comparable complexity. The smallest average for a question or task was 30.4 seconds, and the largest was 65.5, with a SD of 10.1.

The following sections examine subsets of the data sets. We apply standard statistical tests to check our hypotheses. For the following analysis, the entire data is analyzed, including the 'With UML' and 'Without UML' data sets.

TABLE 4. SUMMARY OF QUESTIONS AVERAGES

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	T1	T2	T3	T4
W/O UML	41	43	29	30	39	53	54	56	40	48	66	69
W/ UML	45	55	32	35	40	53	41	47	50	68	40	62
AV	43	49	30	33	39	53	47	51	45	53	53	66

A. *Examining Data for C++ and Java*

The objective of this analysis is to test if there is a statistically significant difference between the data sets for both C++ and Java. This is important because if there is, then we should assume that the two data sets come from distinct populations. If not, and this is our hope, then both Java and C++ come from the same population and we can confidently use their data as representation for object-oriented technology.

Using a two-tailed t-test to measure the statistical significance, there is no significant difference in the data sets for the 'Without UML' set (p = 0.92) and 'With UML' data set (p = 0.88).

As confirmatory evidence (in case of significant departure from the normality requirements of the t-test), we also applied the Mann-Whitney test (U-test). We received similar findings. We note here that there is no reason not to assume normality in the case of the data sets for C++ and Java. However, other studies have recommended that normality should be assumed only when the data set is large, and the sample is representative of the entire population [18]. Representations assumptions have not been tested for our sample. Our data sets are not large enough to justify normality assumption.

B. *Examining Data for C++/Java and Alf*

Now that we have confirmed that both C++ and Java data come from the same population, they can be treated as a single data set. This significantly simplifies the analysis. In this section, we analyze the data sets for C++/Java and Alf.

We run a two-tailed t-test to measure the statistical significance between the average of C++ and Java on one side, and Alf on the other side. The test indicates that the data for Alf is significantly lower than that of C++ and Java (p=1.5x10-8). This means that participants took significantly less time to respond to questions when the system is represented using Alf notation.

Similarly, and as confirmatory evidence (in case of significant departure from the normality requirements of the t-test), we also applied the Mann-Whitney test (U-test), Alf's data set is still significantly lower than that of C++ and Java (p = 8.7x10-9) with a W value of 2722. So using this test we also arrive at the same conclusion.

C. *Examining Data for C++/Java and Umple*

We are not expecting to find significant difference in the case of C++/Java and Umple data sets. The descriptive analysis suggests that both Alf and Umple performance were comparable, despite Umple being a little worse that Alf (a standardized language).

Two-tailed t-test to measure the statistical significance between the average of C++ /Java on one side, and Umple, on the other side, indicate that Umple's performance is better. The t-test indicates that the data for Umple is significantly lower than that of C++ and Java (p=1.1x10-8).

The Mann-Whitney test (U-test) indicate that Umple's data set is still significantly lower than that of C++ and Java (p = 9.2x10-7) with a W value of 2073. So using this test we also arrive at the same conclusion.

Therefore, we can reject the null hypothesis, H01 and state that:

H1: A system written in Umple or Alf is more comprehensible than an equivalent implementation of the system in Java or C++.

D. *Examining Data for Alf and Umple*

Using a two-tailed t-test to measure the statistical significance, Alf and Umple do not have significantly different average times (p=0.9). This is true for the 'With UML' and 'Without UML' data sets, and for both sets combined. A Mann-Whitney test (U-test) confirms the same findings (P = 0.07) and a W value of 4612.2.

We have conducted additional tests on the data, which did not conflict with any of our findings. For example, we conducted standard deviation analysis and sign tests analysis [19]. The standard deviation analysis classifies the data points into two categories; one where the data falls within the mean +/- the standard deviation, and the second where the data points falls beyond this range. The concept is that if the data were significantly different than the mean, then a significant percentage would fall beyond the specified range. Our objective was to examine if there is any hidden evidence in the data, especially between Alf and Umple. We also conducted the same tests on subsets of the data. For example, we divided the data based on whether it is a comprehension question or bug fixing data. Our tests and analysis did not suggest any significant difference between Alf and Umple.

E. *Examining Data for 'With UML' and 'Without UML'*

To test the second hypotheses, we analyze the data for 'with UML' and data for 'Without UML' for all artifacts and all participants.

From the results shown in TABLE 3, we do not find any statistically significant difference. The two-tailed t-test does not result in any statistical difference between the 'with UML' and 'Without UML' data sets (P=0.99).

Therefore, we can reject the second hypothesis, and state that:

H2o: UML visual representation does not enhance comprehension when used with action language or object-oriented code.

F. *Discussion*

The main finding of this experiment is that action languages have a significant comprehensibility benefits when compared to OO systems. This is particularly true for highly abstracted systems such as those used in this experiment (i.e. metamodels). Another finding is that the availability of UML models does not seem to have an impact on the comprehension of such systems. We interpret these two key findings as follows.

The comprehension benefits of the action languages code are both significant and consistent. This is to be expected especially for such model-intensive systems. In fact, one can argue that any software system that is large enough will have significant model-like abstractions. The abstractions could be explicit, i.e. represented by UML or an action language, or could be implicitly specified in code.

The presence of UML artifacts did not have a significant effect on the results. In the case of OO languages, we attribute this to the fact that there is a significant representational gap between the UML notation, and its equivalent mapping in C++ and Java. This made participants focus more on the code in answering the questions. However, for the action languages, the interpretation of this result is that those model-based programming languages successfully bridged the gap with UML; hence, the UML notation did not offer much added comprehensibility value.

V. Threats to Validity

Threats to validity of the experiment and how we tried to mitigate them are described in this section.

A. *Presentation Format*

It is possible that the experiment design sidelined the value of the UML notation. We note that UML models were presented as static images. Participants could not interactively navigate the model. On the other hand, participants were more engaged with the code (object-oriented or action languages). This different in presentation may have affected the participants' engagement with the UML models. We tried to mitigate that by managing the complexity of the systems, to reduce the need for interactivity. We also kept the UML diagrams concise and legible.

B. *Number of Participants*

Thirty-two (32) participants is relatively a small number. However, we used statistical analysis on the data and that yielded strong evidence. We also did not notice any significant difference when running parametric (t-test) and non-parametric test (Mann-Whitney test). However, it is still possible that a larger, more representative sample may have yielded different results.

C. *Participant Experience*

Our participants were relatively knowledgeable about object oriented languages and UML. It is possible that their knowledge may have influenced the results of this experiment. To mitigate this risk, we collected profiling information and tested participants' responses against their knowledge. We were not able to find any evidence that more knowledgeable participants answers were different statistically from not-as-knowledgeable participants' responses. We analyzed the data for each of the 16 participants independently and harmonized their results based on their level of experience. We also looked for any possible significant deviation from the entire experiment averages but could not find any. We used the t-test, Mann-Whitney test, as well as the sign test and the standard deviation analysis [19].

Despite the participants were potentially more knowledgeable about UML than the general software engineering community, they had comparably little background on Alf or Umple. None of the participants reported that their previous knowledge in Alf or Umple was higher than C++ or Java. This means that if participants' experiences and knowledge had an effect on the experiment, it would have been to the benefit of OO languages.

D. *Non-Representative Systems*

This is an external validity threat that our systems are not representative of the real software artifacts. We accept this threat, and in fact, our sample systems were more model-intensive than the typical software artifact. Our samples are an incomplete system, taken and modified from a real software artefact (Umple code). We therefore concur with this threat. One should be aware of this threat when generalizing the results of this experiment.

E. *Non-Representative Complexity*

It is also possible that the systems were not complex enough to realize the comprehensibility value of the UML graphical notation, nor the verboseness of the textual notation. Unfortunately, it was hard to assess the required complexity level in this experiment upfront. We considered the UML metamodel that is notoriously known to be complex to be representative. However, a variation of this experiment could be designed with more complex systems.

F. *Question and Tasks Interpretation*

This is an internal validity threat for our experiment. The threat is that participants may have interpreted the questions in a way that affects the experiment results. For example, a participant may have taken more time to comprehend the question or a task, rather than time to reflect on the problem using the notation under the study,. This threat was mitigated by randomly assigning the participants to the different configurations. We also piloted the questions and tasks, and also had three researchers review our questions and tasks to minimize this threat.

G. Use of Pairwise Comparison

We used pairwise comparisons when analyzing our data sets. For example, we separately compared pair of data sets for all of our configurations. We understand that the more we use this type of analysis, the greater the chance of a Type I error (i.e. rejecting the Null hypothesis when it is actually true). Multi-way comparisons are more suitably tested using a test such as ANOVA, especially when there is more than one configuration. However, this approach is only relevant when the P value is close to the significance threshold, and this did not apply to our analysis. Our P values were far from the significance threshold, either being very low or very high. Therefore, we did not see the need to run ANOVA tests.

VI. Related Work

In a prior work, we have investigated conceptional and notational alternatives related to the design and implementation of Action Languages [35] [37]. One key contribution of this work is a bottom-up Action Language design approach to facilitate language adoption and improve notation comprehension. In another prior work, we have investigated the challenges for empirical studies of software engineering tools and technologies at different stages of maturity [27]. We find the most challenging studies are those that attempt to evaluate tools, approaches, or notations, prior to any wide adoption. The study reported in this paper falls into this category. Action languages are nowhere near consistent and wide adoption by professionals.

The literature however has many works reported on empirical evaluations of different notations [36]. Hendrix evaluated the comprehension level of code control structures by also measuring the time span the participants took to answer comprehension questions [20]. This is similar to the approach adopted in our experiment. Briand el al. [21] evaluated two different ways of presenting information. They found no evidence that “good structured design is easier to understand than bad structured design”. Gemino and Wand investigated the use of mandatory subtypes versus optional properties in entity-relationship model (ERM) [33]. Similar to our study, they created two equivalent models and measured participants’ comprehension. They conclude that mandatory relationships lead to improved comprehensibility despite apparent increase in model complexity.

Rather than focusing on comprehension, usability studies focuses on the ease of manipulation and interaction with a tool or a notation. Hornbæk investigated current practices and challenges in conducting usability studies [34]. David Chin [24] has investigated the usability of system models and user models. In his study, he also finds little empirical investigations of the usability of models. In this work, he provides rules of thumb for experimental design, useful tests for covariates, and common threats to experimental validity. Chin also proposed reporting standards including effect size and power, which we have adopted to a large extent in this experiment.

VII. Conclusion and Future Work

In this work, we compared the newly emerging UML action languages with the more established object-oriented languages in terms of comprehensibility. Through a controlled experiment, we found the former to be much more comprehensible than the latter, judged by the time it took participants to answer comprehension and bug fixing questions on two different software systems. We also assessed whether having access to UML notation beside the either object-oriented or action language would result in added benefits to comprehension. However, we did not notice any significant impact in this experiment. We explain this for action languages by the fact that their code is already at the model level. However, it was surprising for the object-oriented code case. We offered insights into the results and outlined possible threads to validity.

We further note that we did not analyze the comprehension questions separately from the bug fixing ones. In other words, we did not explore whether there is any significant difference if the data was sliced along the category of the question. We leave this analysis to future work. We also did not analyze how participants arrived at their answers. We do not know whether participants have used the UML models only, the code only, or both, to answer questions. This particular analysis is also left to future work.

References

[1] OMG (2015) Action Language for Foundational UML (Alf), Concrete Syntax for a UML Action Language. Available: http://www.omg.org/spec/ALF/

[2] Mellor, Stephen J., et al. "An action language for UML: proposal for a precise execution semantics." The Unified Modeling Language.«UML»’98: Beyond the Notation. Springer Berlin Heidelberg, 1999. 307-318.

[3] Sunyé, Gerson, et al. "Using UML action semantics for executable modeling and beyond." Advanced Information Systems Engineering. Springer Berlin Heidelberg, 2001.

[4] Purchase, Helen C., et al. "Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study." Proceedings of the 2001 Asia-Pacific symposium on Information visualisation-Volume 9. Australian Computer Society, Inc., 2001.

[5] Purchase, Helen C., et al. "UML class diagram syntax: an empirical study of comprehension." Proceedings of the 2001 Asia-Pacific symposium on Information visualisation-Volume 9. Australian Computer Society, Inc., 2001.

[6] Timothy C. Lethbridge, Andrew Forward, Omar Badreddin. Problems and Opportunities for Model-Centric vs. Code-Centric Development: A Survey of Software Professionals, in the proceedings of C2M:EEMDD 2010.

[7] Büttner, Fabian, and Martin Gogolla. "Modular embedding of the object constraint language into a programming language." Formal Methods, Foundations and Applications. Springer Berlin Heidelberg, 2011. 124-139.

[8] Rose, Louis M., et al. "Constructing models with the human-usable textual notation." Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, 2008. 249-263.

[9] Object Management Group (OMG). " Concrete Syntax for a UML Action Language RFP", accessed 2012, http://www.omg.org/cgi-bin/doc?ad/2008-9-9.

[10] Planas, Elena, et al. "Alf-Verifier: an eclipse plugin for verifying Alf/UML executable models." Advances in Conceptual Modeling, 2012. Springer Berlin Heidelberg, 2012.378-382.

[11] Chaves, R. " TextUML", accessed 2014, http://abstratt.com/

[12] Perseil, Isabelle. "ALF formal." Innovations in Systems and Software Engineering 7.4 (2011): 325-326.

[13] Badreddin, Omar. "Umple: a model-oriented programming language." Software Engineering, 2010 ACM/IEEE 32nd International Conference on. Vol. 2. IEEE, 2010.

[14] Object Management Group (OMG), Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), available: http://www.omg.org/spec/ALF/1.0.1.

[15] Mellor, Stephen J., and Marc J. Balcer. Executable UML: a foundation for model-driven architecture. Addison-Wesley Professional, 2002.

[16] Dzidek, Wojciech J., Erik Arisholm, and Lionel C. Briand. "A realistic empirical evaluation of the costs and benefits of UML in software maintenance." Software Engineering, IEEE Transactions on 34.3 (2008): 407-432.

[17] "Umple language online." accessed 2015, www.try.umple.org.

[18] Jarque, Carlos M., and Anil K. Bera. "Efficient tests for normality, homoscedasticity and serial independence of regression residuals." Economics Letters 6.3 (1980): 255-259.

[19] S. Mohammad. "From once upon a time to happily ever after: Tracking emotions in novels and fairy tales". 2011. ACL HLT 2011pp. 105.

[20] D. Hendrix, J. H. Cross II and S. Maghsoodloo. "The effectiveness of control structure diagrams in source code comprehension activities". 2002. IEEE Trans.Software Eng.pp. 463-477.

[21] L. C. Briand, C. Bunse, J. W. Daly and C. Differding. "An experimental comparison of the maintainability of object-oriented and structured design documents". 1997. Empirical Software Engineering vol 2, pp.291-312.

[22] Friedenthal, Sanford, Alan Moore, and Rick Steiner. A practical guide to SysML: the systems modeling language. Access Online via Elsevier, 2011.

[23] Omar Badreddin. Model Orientation Experiment Specification. Accessed 2014. Available: http://obahy.files.wordpress.com/2014/02/experiment-specification.docx.

[24] Chin, David N. "Empirical evaluation of user models and user-adapted systems." User modeling and user-adapted interaction 11.1-2 (2001): 181-194.

[25] Badreddin, Omar, and Timothy C. Lethbridge. "Model oriented programming: Bridging the code-model divide." *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*. IEEE, 2013.

[26] Badreddin, Omar Bahy, Andrew Forward, and Timothy C. Lethbridge. "Model oriented programming: an empirical study of comprehension." *CASCON*. 2012.

[27] Badreddin, Omar. "Empirical evaluation of research prototypes at variable stages of maturity." *User Evaluations for Software Engineering Researchers (USER), 2013 2nd International Workshop on*. IEEE, 2013.

[28] Rumpe, Bernhard. "Executable Modeling with UML. A Vision or a Nightmare?." arXiv preprint arXiv:1409.6597 (2014).

[29] Schamai, Wladimir, Peter Fritzson, and Chris JJ Paredis. "Translation of UML state machines to Modelica: Handling semantic issues." Simulation (2013): 0037549712470296.

[30] Planas, Elena, et al. "Alf-Verifier: an eclipse plugin for verifying Alf/UML executable models." Advances in Conceptual Modeling. Springer Berlin Heidelberg, 2012. 378-382.

[31] Jedlitschka, Andreas, Marcus Ciolkowski, and Dietmar Pfahl. "Reporting experiments in software engineering." Guide to advanced empirical software engineering. Springer London, 2008. 201-228.

[32] Lazăr, C. L., I. Lazăr, B. Pârv, S. Motogna, and I. G. Czibula. "Tool Support for fUML Models." Int. J. of Computers, Communications & Control 5, no. 5 (2010): 775-782.

[33] Gemino, Andrew, and Yair Wand. "Complexity and clarity in conceptual modeling: comparison of mandatory and optional properties." *Data & Knowledge Engineering* 55.3 (2005): 301-326.

[34] Hornbæk, Kasper. "Current practice in measuring usability: Challenges to usability studies and research." *International journal of human-computer studies*64.2 (2006): 79-102.

[35] Badreddin, Omar, Timothy C. Lethbridge, and Andrew Forward. "Investigation and evaluation of UML Action Languages." *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*. IEEE, 2014.

[36] Burton-Jones, Andrew, and Peter N. Meso. "Conceptualizing systems for understanding: an empirical test of decomposition principles in object-oriented analysis." Information Systems Research 17.1 (2006): 38-60.

[37] Burton-Jones, Andrew, and Peter Meso. "The effects of decomposition quality and multiple forms of information on novices' understanding of a domain from a conceptual model." Journal of the Association for Information Systems 9.12 (2008): 1.