Comparative Case Studies in Agile Model-Driven Development

K. Lano[1], H. Alfraihi[1,3], S. Kolahdouz-Rahimi[2], M. Sharbaf[2], and H. Haughton[1]
{hessa.alfraihi,kevin.lano}@kcl.ac.uk
{sh.rahimi,m.sharbaf}@eng.ui.ac.ir

[1] Dept. of Informatics, King's College London, London, UK
[2] Dept. of Software Engineering, University of Isfahan, Isfahan, Iran
[3] Dept. of Information Systems, Princess Nourah bint Abdulrahman University, Saudi Arabia

Abstract. This paper reports on experiences of integrating Agile and Model-Driven Development, for the development of code generators and financial systems. We evaluate the benefits of the Agile MDD approach by comparing Agile non-MDD and Agile MDD developments of code generators, and an agile MDD development of a financial application with three other independent versions of the same application developed using different approaches. We also compare the functionality of the systems and a variety of technical debt metrics measuring the quality of the code and its design. Based on the case study results, we have found evidence that the use of Agile MDD leads to reductions in development effort, and to improvements in software quality and efficiency.

1 Introduction

Integration of Agile and Model-driven Development (Agile MDD) is of significant interest to practitioners who want to utilise the benefits of both approaches. However, only limited research has been carried out to investigate the impact of their integration [1,3]. In this paper we compare Agile MDD and other approaches in two different application areas: (i) code generators and (ii) financial systems. Agile MDD developments of UML to C and UML to Python code generators are compared in Section 3 to Agile non-MDD developments of other code generators. In Section 4 a comparison between four independent developments of the same finance application is carried out, with the developments using different approaches: Agile MDD, MDD, Agile, and non-Agile hand-coded development.

2 Research methodology

Our research goal is to *evaluate the impact of integrating Agile development and MDD*. For this goal we ask the following research questions:
RQ1: What is the impact of integrating Agile development and MDD on the

software product?
RQ2: What is the impact of integrating Agile development and MDD on the software development process?

In order to answer RQ1, the properties of quality, efficiency and maintainability of the case study versions were compared. For quality and maintainability, we use measures of technical debt [9]: **EAS** (Excessive application complexity c [7] or LOC, with a threshold of 1000); **ENR** (Excessive number of rules, *nrules* > 10); **ENO** (Excessive number of operations/methods *nops* > 10); **EHS** (Excessive operation/method size, threshold of 100 for c or LOC); **MHS** (Maximum operation size); **ERS** (Excessive rule size $c > 100$); **CC** (Cyclomatic complexity of rule logic or of procedural code > 10); **MCC** (maximum CC); **CBR** (Number of rule/operation explicit or implicit calling relations $>$ *nrules* + *nops*, or any cyclic dependencies exist in the rule/operation call graph); **EPL** (number of parameters for a rule or operation/method > 10); **DC** (duplicate expressions or statements with token count > 10).

We use threshold values based on those used in the PMD technical debt analysis tools (pmd.github.io): EAS 1000 LOC; EHS 100 LOC; EPL 10 parameters; CC 10; ENO 10 per class.

3 Code generation case study

As part of the UML-RSDS toolset (www.nms.kcl.ac.uk/kevin.lano/uml2web) a number of code generators are provided to map UML and OCL to program code. Translators to Java (3 versions), C# and C++ were manually coded in Java using an agile approach over the period 2003 to 2015, whilst more recent code generators, UML2C for ANSI C and UML2Python for Python, have been developed using our Agile MDD process [2] in UML-RSDS [8]. Specifically, the code generators were developed by writing UML-RSDS specifications, from which code is automatically generated both for testing purposes and for delivery of executables. All of the generators were written by the same single developer (the first author). The source metamodel of UML2C and UML2Python consists of 33 classes (www.nms.kcl.ac.uk/kevin.lano/libraries/design.km3), whilst the C target metamodel has 24 classes. UML2C and UML2Python are the largest applications that have been developed using UML-RSDS.

A systematic requirements engineering process was followed for the UML2C and UML2Python generators, involving document mining, scenario analysis, goal decomposition, exploratory and evolutionary prototyping, inspection and refactoring.

For UML2C, the development was organised into 5 iterations, one iteration for each of the following main functional requirements:

- F1.1: Translation of types
- F1.2: Translation of class diagrams
- F1.3: Translation of OCL expressions
- F1.4: Translation of activities

– F1.5: Translation of use cases.

In each iteration an informal mapping of language elements from UML to the target languages was constructed. For example, Table 1 shows the mapping of types from UML to C for iteration 1.

Scenario	*UML element* e	*C representation* e'
F1.1.1.1	*String* type	`char*`
F1.1.1.2	int, long, double types	same-named C types
F1.1.1.3	boolean type	`unsigned char`
F1.1.2	Enumeration type	C `enum`
F1.1.3	Entity type E	`struct E*` type
F1.1.4.1	*Set*(E) type	`struct E**` (array of E', without duplicates)
F1.1.4.2	*Sequence*(E) type	`struct E**` (array of E', possibly with duplicates)

Table 1: Informal mapping scenarios for UML types to ANSI C

Table 2 shows the corresponding mapping for Python. There were 2 iterations with F1.1 and F1.2 considered in the first, and F1.3, F1.4, F1.5 in the second.

Scenario	*UML element* e	*Python representation* e'
F1.1.1.1	*String* type	`str`
F1.1.1.2	int, long, double types	int, int, float
F1.1.1.3	boolean type	`bool`
F1.1.2	Enumeration type	`Enum` class instance
F1.1.3	Entity type E	class `E`
F1.1.4.1	*Set*(E) type	`set(E)`
F1.1.4.2	*Sequence*(E) type	`list(E)`

Table 2: Informal mapping scenarios for UML types to Python

In contrast to the large effort expended on writing code in the case of the manually-coded code generators, the principal effort in the UML2C and UML2Python developments was the formal specification definition and review/testing of this specification. In UML-RSDS, the construction, review, testing, optimisation, correction and refactoring of the application specification replaces the corresponding code construction processes in a conventional non-MDD approach. No manual coding was used for either UML2C or UML2Python, and a declarative/functional style of specification was used for both.

The formal specification consists of a class diagram and use cases, defined by OCL constraints. An example constraint from UML2C is the following, which expresses formally the informal mapping F1.1.3:

```
Entity::
```

```
CPointerType->exists( p | p.ctypeId = typeId &
   CStruct->exists( c | c.name = name & c.ctypeId = name &
                        p.pointsTo = c ) )
```

The constraints map instances of the UML metamodel into instances of a C metamodel. Additional constraints define how concrete C text is produced from the C model. In the case of UML2Python, text was directly produced from the UML model.

The design structure of the UML2C synthesiser is an external transformation chain of two transformations: *uml2Ca*, which produces the app.h header file for the C output, and *uml2Cb*, which produces the app.c main code file. F1.1 and F1.2 are implemented in *uml2Ca* (an internal chain of 2 subtransformations), and F1.3, F1.4 and F1.5 are implemented in *uml2Cb* as a chain of 3 subtransformations. For UML2Python a single *uml2py* transformation was delivered, with a hierarchical organisation: *model2py* (implementing F1.1, F1.2, F1.5) is a client to *stat2py* (F1.4), which is a client of *exp2py* (F1.3).

3.1 Evaluation

We compare UML2C and UML2Python to the manually-coded C++ code generator. The developer had low experience in C++, C and Python, and hence the three generators are comparable in each requiring significant background research and experimentation with different generation strategies. UML2C was the first generator to target a procedural language, and presented greater conceptual difficulties than previous generators (eg., how to represent inheritance). UML2Python was the first generator for a dynamically-typed language.

Table 3 compares the overall development effort in person days between the C++, C and Python code generators.

Stage	*C++ generator*	*UML2C*	*UML2Python*
Requirements Elicitation	10	17	2
Evaluation/Negotiation	10	5	1
Specification	60	56	2
Review/Validation	50	57	1
Implementation/ Testing	180	49	3
Total	310	184	9

Table 3: Development effort for C++, C and Python code generators

A major factor in this difference is the much simpler and more concise transformation specification of the C and Python code generators (expressed in UML-RSDS) compared to the Java code of the C++ code generator. Not only is the UML2C specification 7 times shorter than the UML to C++ Java code, but the latter is scattered over multiple source files (eg., Attribute.java, Association.java,

Entity.java, etc) and mixed with the code of other parts of UML-RSDS, making debugging and maintenance more complex compared to the C translator, which is defined in 2 self-contained specification files. In total, the core code of the UML-RSDS tools is 90,500 lines of Java code, of which approximately 20% (18,100 lines) is the C++ code generator. In contrast the UML2C specification is 874 (uml2Ca) and 1576 (uml2Cb) lines, in total 2450 lines. The UML2Python generator is a single file of 605 lines. The OCL specification of UML2C is highly declarative and corresponds directly to the informal requirements, hence it is easier to understand and modify compared to a programming language implementation. UML2Python reused the structure of the UML2C informal specification, and substantial parts of the UML2C formal specification, with simplifications due to the higher level of the Python language compared to C.

Whilst UML2C is explicitly divided into 5 sequential stages, each subdivided into model to model and model to text modules, the C++ generator has a monolithic structure. UML2Python has a hierarchical structure of 3 strongly separated subsystems. Only two design patterns (Iterator and Visitor) are used in the C++ generator, whilst 13 are used to organise UML2C and 5 in UML2Python.

Table 4 summarises the differences in software quality aspects between the C++ generator and UML2C and UML2Python.

Measure	*C++ generator*	*UML2C*	*UML2Python*
Size (LOC)	18,100	2,450	605
Abstraction level	Low (code)	High (specification)	High (specification)
Software architecture	Partial	Detailed	Detailed
Modularity	Low (1 module)	High (10 modules)	Medium (3 modules)
Cohesion	Low	High	High
Coupling	Low	Low	Low
Design patterns	2	13	5

Table 4: Software quality aspects of C++ and C, Python code generators

We can also compare the level of design flaws or technical debt in the C++ translator and in UML2C and UML2Python. For the C++ translator the data has been calculated using the PMD tool (https://pmd.github.io). For UML2C and UML2Python we used the technical debt metrics built in to UML-RSDS.

Measures of fan-out, duplicate code and coupling between operations are not measured by PMD, so are omitted in the comparison. Table 5 compares the TD figures for UML2C, UML2Python and the C++ generator. For the latter, the figures for EHS/ERS, CC and EPL are produced by dividing the total numbers of flawed operations in the entire UML-RSDS toolset by 5. For EAS and ENO+ENR we count the number of classes (C++ generator) or sub-transformations (UML2C and UML2Python) with these flaws. It can be seen that the figures for UML2C and UML2Python are generally lower than for the C++ translator.

Transformation	EAS	$EHS + ERS$	CC	$ENO + ENR$	EPL
C++ generator	5	16.6	110.6	28	0.4
UML2C	2	13	32	4	0
UML2Python	1	8	6	3	0

Table 5: C++ translator versus UML2C and UML2Python technical debt

We also compared the efficiency of the code generators applied to example models of varying size up to 100 classes with 100 features in each class.

4 Financial case study

In this case study we compared the Agile MDD approach with three other approaches: MDD, Agile, and hand-coded approaches. The four applications all implemented the same problem and were completed independently using different development approaches by different teams. The case study problem is to calculate and evaluate the risk of financial instruments known as *Collateralized Debt Obligation*s (CDOs) [4,5]. Risk analysis of a CDO involves calculating the probability $P(S = s)$ of a total credit loss s from the CDO. The specification of the calculation is given in [5].

To evaluate efficiency, we measured the total execution time of each application version on a sample dataset input of 16 sectors and output of P(S = s) for all values of s $\leq$ 20.

In order to answer RQ2, we used qualitative measures through an opinion survey, in which the developers were asked to report the main benefits they perceived, along with the issues they faced in each approach.

Agile MDD approach The CDO application was implemented using the UML-RSDS Agile MDD approach [8,2]. The customer of this application was a financial analyst working in a financial company. The developer had 10 years experience of UML-RSDS and had no prior experience in financial applications.

MDD approach The CDO case study was redeveloped using the same specification of [5]. The developer used EMF/ECORE [10] to specify the metamodels, and the transformation was implemented using the Epsilon Transformation Language (ETL) [6]. This solution was developed by one developer who had 4 years experience of ETL and had no experience in financial applications.

Agile approach The CDO application was redeveloped using the Scrum process. The application was implemented in Java and the development process was organised in three iterations (each one-week long). Some Scrum techniques were used such as product backlog, sprint backlog, user stories, requirements prioritisation, sprint planning, sprint review, and frequent customer involvement. The developer had 5 years experience of Java programming and had no experience

in financial applications.

Original CDO application The CDO application was previously developed in C++ and used in a financial company. It was developed using a traditional code-centric non-agile approach. The developer had over 20 years experience in C++ programming and was a financial analyst in that company.

Table 6 presents the execution time and the size in LOC of the four application versions

	Agile MDD	MDD	Agile	Original
Execution Time of P(S=s) for all s $\leq$ 20	23ms	123ms	39ms	231ms
Execution Time of P(S=s) for all s $\leq$ 50	93ms	531ms	> 15min	> 15min
Size(LOC)	94	143	196	163

Table 6: Execution time and LOC for each CDO version

For RQ2 we use a 3-point scale Low, Medium, High to quantify the different aspects reported by developers (Table 7).

Aspect	*Agile MDD*	*MDD*	*Agile*	*Original*
Development effort	Medium, due to small specification size	High, due to need to rework specification	Medium	High
Errors in deliverables	Low due to direct customer feedback	High: Missing functionality	Low, due to customer feedback	Low
Ability to make changes	High, due to small/declarative specification	High, due to small specification	Medium	Medium
Development organisation	High: based on iterations	Medium	High: based on iterations	Medium

Table 7: Impact on development process (CDO case)

Direct communication with the customer representative during development was very important in reducing errors and correcting misunderstandings in the Agile and Agile MDD cases. The major challenge for the developers of the Agile MDD, MDD and Agile versions was in understanding the domain and formalising the required functionality correctly.

5 Discussion

5.1 Code generator case study

For the code generator versions, there were no significant differences in efficiency between the Agile MDD C and Python code generators and the Agile non-MDD C++ code generator. In terms of product quality, there were clear gains in reduced application size and reduced technical debt, and improved system structuring. In turn, these benefits make the ongoing maintenance of the UML2C and UML2Python code generators more effective than for the manually-coded generators. For example, we recently implemented a source metamodel enhancement within 30 minutes for each of these generators, the corresponding change for the C++ generator was around 2 hours. Lessons learnt from the UML2C development were applied to UML2Python, resulting in a much reduced development effort (5% of the UML2C effort) in the case of UML2Python. Another factor here is that the language gap between UML and Python is much smaller than between UML and C (however the gap between UML and C is higher than between UML and C++). Substantial reuse of the UML2C specification in UML2Python was facilitated by the concise and semantically-transparent nature of this specification. Table 8 summarises the difference in TD levels between the three code generator versions.

Transformation	flaws	flaws/LOC
C++ generator	155.6	0.009
UML2C	51	0.021
UML2Python	18	0.029

Table 8: Summary of code generator technical debt

Although the flaw density is lower for the non-MDD approach, the number of flaws is much higher. The overall functionality of the C++ and C code generators is comparable, yet the C++ generator has 3 times the number of flaws of the C generator.

5.2 Financial case study

For the financial case study, in terms of efficiency, the application developed using Agile MDD was the fastest. With regard to the product quality, the Agile MDD application had consistently better metrics than the other versions.

Table 9 and Table 10 present the average of the values of the metrics for the two Agile versus the two non-Agile approaches and for the two MDD approaches versus the two non-MDD approaches. Agile approaches therefore have better values than non-Agile approaches in 8 of the 9 measures while MDD approaches have better values than non-MDD approaches in 7 of the 9 measures.

	Effic.	LOC	ENO	MHS	MCC	CBR	DC	flaws	flaws/LOC
Agile	31ms	145	10.5	15	4	9(1.5)	1	3	0.0207
Non-agile	177ms	136	11	26	4.5	12(1.5)	1.5	4	0.0294

Table 9: Agile versus non-Agile approaches for CDO

	Effic.	LOC	ENO	MHS	MCC	CBR	DC	flaws	flaws/LOC
MDD	73ms	118.5	10	13.5	2.5	12(2)	0.5	3	0.0253
Non-MDD	135ms	162.5	11.5	27.5	6	9(1)	2	4	0.0246

Table 10: MDD versus non-MDD approaches for CDO

5.3 Outcome of research questions

For $RQ1$, we found improvements in the quality aspects and TD measures for the Agile MDD solutions in both case studies, and in efficiency of the financial case.

For $RQ2$ we found a 40% reduction in development effort for UML2C compared to the C++ code generator, despite the greater challenges of generating C compared to C++. Specification-level reuse was highly beneficial in the UML2Python case, whilst only code-level reuse was possible for the C++ generator.

Errors in deliverables were reduced in the Agile MDD and Agile versions of the CDO case due to direct customer collaboration in development. In both the code generator and CDO case studies, the use of incremental development and iterations helped in the organisation and decomposition of the Agile MDD developments.

5.4 Threats to validity

A significant issue concerns the development team size. Agile methods emphasise communication and team collaboration. In this research, the code generator application versions and MDD and original CDO versions were implemented by single developers, whilst the Agile MDD and Agile CDO versions involved a main developer and an advising customer representative. Therefore it is difficult to generalise our conclusions for larger projects.

Secondly, only one form of Agile MDD is used (UML-RSDS) and only one form of Agile (Scrum), so our analysis does not necessarily hold for other varieties of these approaches.

Conclusion

Our results in these case studies show some evidence that Agile MDD has improved efficiency and quality of delivered software. In future work, we intend to extend this study using larger case studies with larger development teams.

References

1. Hessa Alfraihi and Kevin Lano. The integration of agile development and model driven development: A systematic literature review. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, 2017.
2. Hessa Alfraihi and Kevin Lano. A process for integrating agile software development and model-driven development. In *In 3rd Flexible MDE Workshop (FlexMDE) co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2017)*, CEUR Workshop Proceedings, pages 412–417, Austin, TX, USA, 2017. CEUR-WS.org.
3. Håkan Burden, Sebastian Hansson, and Yu Zhao. How MAD are we? Empirical Evidence for Model-driven Agile Development. In *Proceedings of XM 2014, 3rd Extreme Modeling Workshop*, volume 1239, pages 2–11, Valencia, Spain, September 2014. CEUR.
4. M Davis, V Lo, et al. Infectious defaults. *Quantitative Finance*, 1(4):382–387, 2001.
5. Ola Hammarlid et al. Aggregating sectors in the infectious defaults model. *Quantitative Finance*, 4(1):64–69, 2004.
6. Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.
7. K. Lano, S. Kolahdouz-Rahimi, M. Sharbaf, and H. Alfraih. Technical debt in model transformation specifications. In *ICMT*, 2018.
8. Kevin Lano. *Agile model-based development using UML-RSDS*. Boca Raton: CRC Press, 2017.
9. Radu Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9–1, 2012.
10. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2 edition, 2008.