ON THE EVOLUTION OF IMEDIA IMPLEMENTATIONS[*]

T. CLEENEWERCK AND D. DERIDDER AND J. BRICHAU AND T. D'HONDT
Vrije Universiteit Brussel, PROG
Pleinlaan 2, 1050 Brussels
Belgium, BE
E-mail: {tcleenew|dderidde|jbrichau|tjdhondt}@vub.ac.be

With the advent of iMedia, the traditional content component was extended with a behavioral component (i.e. software). The development of this behavioral component using traditional software development techniques is cumbersome because of the extreme deadlines and extremely short time-to-market situation. We propose a new development approach that provides the media producer with sufficient control to define and change the product in very short time frames. The system is based on existing technologies like generative programming, transformation systems and domain engineering. Since the iMedia domain is in continuous flux, and these technologies are mostly designed for stable domains, the evolution of the implementation was a crucial problem hampering its successful application. Solutions and mechanisms are presented that ameliorate the modularity and consequently the evolution.

1. Introduction

Media broadcasting companies are currently augmenting their media-offer with different kinds of interactivity such as online gaming, virtual community building, and active TV show participation at home. The production of this interactive form of media (iMedia) encompasses the publication of a behavioral component (the software that provides the interactivity), along with the original content component. The development and evolution of this behavioral component by using traditional software development techniques is cumbersome. Hence there exists an urgent need for a different approach to handle this kind of development. This is easily motivated when looking at the specific characteristics of the iMedia development context.

iMedia software development takes place in an environment in which *extreme strict deadlines* and an *extremely short time-to-market* situation constrain the development process. As broadcasting occurs in real-time, missing the broadcast deadline consequently renders the iMedia software completely useless. Moreover last minute changes are paramount, since a lot of product features are crystallized as development moves onwards. Traditional software development approaches have trouble accommodating this kind of environment.

[*] This research is partially performed in the context of the e-VRT Advanced Media project.

The iMedia Software Generation System (IMSGS) approach we propose in Section 2 combines existing research from the areas of generative programming, transformation systems and domain engineering. In essence the main goal of our approach boils down to providing more autonomy and flexibility to the media producer for adapting the iMedia software product.

Of course from time to time the media producers will require new features that were not anticipated in the original design of the IMSGS. In that case the generation system itself should be changed by a software expert. As reported extensively in literature, it isn't trivial to evolve a system that is based on transformation systems and generative programming technology. So in order to increase the practical feasibility of our approach we have investigated how we could make the evolution of the generation system easier to do (Section 3).

2. An iMedia Software Generation System

An IMSGS is always installed for a certain product range and enables the easy specification and generation of different tailor-made "instances" of this product range. The tailorisation of such an instance is done by the media producer (the domain expert). This stands in shrill contrast to the traditional situation in which the domain expert only plays a prominent role in the early phases of the process. So we are partially transferring the responsibility of adapting the software from the programmer to the domain expert (media producer). This requires advanced software development techniques accompanied by an appropriate software development process. In our work we focus on the former. A good candidate for the latter are agile software development processes, e.g. eXtreme Programming [1], and Adaptive Software Development [7].

2.1. *Domain-specific Languages*

Central to the IMSGS approach is the use and development of *domain-specific languages* (DSLs) [9]. DSLs are languages specifically designed to express a range of applications in a particular domain. This entails that the language constructs of a DSL reflect the concepts of a domain and protects the DSL programmer from non-iMedia-specific technical issues. Therefore, we propose to use DSLs as a means to develop the iMedia software such that the media producers themselves can write, adapt, and maintain the iMedia software. Media producers are thus less dependent on the software developers.

The main role of the software developers in our approach will be to develop these domain-specific languages. The design and implementation of a DSL involves two mappings. The first mapping establishes the concepts of the domain analysis and maps them onto appropriate language constructs. In essence, this mapping is about the design of the language based on a domain analysis. A second mapping establishes a link between the domain language and a general-purpose programming language. In essence, this involves the implementation of a compiler for the DSL.

The design of a DSL should be based on a careful domain analysis. Programs expressed in an iMedia DSL should reflect domain concepts and relations explicitly, allowing a media producer to understand, write and maintain an iMedia DSL program. In contrast, an iMedia program remains an executable specification. This means that on the one hand, the DSL should not only cover the required domain concepts but also more general concepts such as control flow of the program. On the other hand some domain concepts and relations are not explicitly reflected in language constructs of the DSL because their main purpose was to explain the domain. Often, these concepts and relations are implicitly present in the implementation of the DSL compiler. In our approach we make these concepts explicit in a Domain Ontology.

DSL compilers are commonly implemented using generative programming systems such as transformation systems, but also involve the creation of traditional frameworks, components, etc. In general, a program written in a DSL is transformed into a program written in a general-purpose language. The transformation process itself is achieved by programs written in transformation systems and other generative technologies ([8], [10], [3], …). This allows us to deal easily with the inherent large number of possible programs that can be written in a DSL. Note that the output of a DSL compiler is often code that instantiates a framework or that acts as glue code between components.

2.2. *Composition of Domain-specific Languages*

Creating a “universal” DSL in which one could express all possible conceptions of iMedia products is not possible or desirable. Since the resulting complexity of such a DSL, it would also be overly general and thus resemble a general-purpose programming language. In essence, a DSL is developed for a specific kind or aspect of iMedia products, such as one DSL for quizzes, one for online communities, etc. However, it is also the case that many DSLs will have to share common domain concepts, such as layout, user-interface, and communication. Each of these common domain concept groups constitutes a domain on their own, requiring the development of a separate DSL. Consequently, an iMedia product will be specified in multiple DSLs, each addressing a particular aspect of the domain concepts required in the iMedia product. On the technical level, this implies that a compiler of an iMedia DSL is actually a composition of multiple DSL compilers. There will exist a compiler for each domain that should be covered by the “main” DSL. As a result we will have to deal with combining the collection of programs generated by each DSL compiler. This combination is achieved using a compositional generative technology. The difference between compositional generators and transformational generators (written using a transformation system) is that the former generate programs by glueing smaller program parts together via a composition mechanism, while the latter transforms a program into another.

2.3. *Illustration*

In what follows we will briefly illustrate the IMSGS approach with TV Quizz example. To support the media producer in developing his particular iMedia Quiz-software he is provided with a suitable Quiz-DSL based on a Quiz domain ontology, with a compiler to transform the quiz-DSL description into an executable software program. The quiz-DSLs contain several domain concepts (e.g. round, question), but also comprises concepts necessary for describing execution-oriented concepts (e.g. control flow). Note that the actual quiz description will also refer to other aspects of the quiz-product than the ones shown here (e.g. the user interface is described in a specialized UI-DSL).

The next step in the development process is to compile this specification in order to generate the corresponding implementation. Note that the generated product is fully functional and does no longer involve manual programming. Consequently if the producer decides a few hours before broadcasting that the software should be changed, he can easily change the specification and recompile it. For example changing the order in which questions are asked from random to sequential can be easily done. So instead of having to contact the IT-department and file a change request, the media producer now has the autonomy and flexibility of adapting the “software” himself.

Of course this was the case because the original domain analysis of the Quiz-IMSGS anticipated this kind of change. Hence they had already created the necessary DSL language constructs. It is clear that a certain point in time, the producer's needs will no longer match the capacities of the Quiz-IMSGS.

3. Evolution of an IMSGS Implementation

Even though the construction and implementation of a DSL-based system is already greatly facilitated, evolving such a system is still a complex undertaking. This is probably why DSL technology is mostly used in relatively stable domains that have matured over the years (e.g. LaTeX for typesetting). Unfortunately, the domain of media is in continuous flux, exploring new possibilities and trying to exploit advances in available technology. Hence in our research we focus especially on the evolvability of such IMSGS implementations. In the following section we illustrate the impact of unanticipated changes in an IMSGS with the quiz example. In Section 3.2 we will briefly discuss the problem of implicit dependencies which are actually the reason behind the difficulties encountered when evolving an IMSGS. Consequently we will introduce the mechanisms we had to conceive and develop to counter these difficulties in Section 3.3.

3.1. *Impact of Unanticipated Changes*

Let us return to the quiz example from Section 2.3. Suppose a media producer wants to create a quiz with multiple players. Suppose also that the IT-department did not anticipate this in the original quiz-DSL. As a result the Quiz-IMSGS needs to be altered. Unfortunately, adding multiplayer support has severe repercussions on different parts of the IMSGS: the domain ontology, the language constructs, the generators, and the traditional software components.

In the domain ontology we should add concepts such as time, players. We will also have to update existing concepts such as flow, and points to take the players into account. In the flow of a quiz the order in which the player plays and the questions they get to answer must be specified. The point system must be refined to assign points to different players. The semantics defined in the point system and the turns of the players determines the number of times the same question may be answered. These changes must be reflected by concise modifications to the DSL language constructs. First there are a couple of new language constructs needed: a construct to describe and initialize the points, a construct stating the turn of the two players (in this case simultaneous). Second existing language constructs must be changed. The conditions of the points are extended to a Boolean expression over the players that answered the questions. Third the impact changes of language constructs on other constructs must be considered and clarified. On the domain level, there is a relation between the point system, the turn of the players and the number of times the same question may be asked. Clearly this must also be the case in the DSL. Adding multiplayer functionality involved thus a major change in the language semantics. Finally the DSL compiler and the underlying component system must be adjusted, refactored and tested. Naturally this must not corrupt the existing implementation to avoid unexpected changes in the existing iMedia products.

3.2. *Implicit dependencies*

Changing the IMSGS must be reflected in its three main parts: domain models, DSLs and component libraries and frameworks. Since these parts are mapped onto each other by two mappings (section 2), the most difficult step in applying the changes is the co-evolution of those mappings.

It requires traversing and overcoming two mappings made during the initial construction of the DSL: domain concepts to DSL concepts and DSL concepts to components. Each of the mappings is non-trivial and if a change is not applied with care, the internal correctness and consistency of the DSL can be broken. Currently there is little or no support to keep the mapping between domain concepts to DSL concepts alive, traceable and manageable. This renders the dependencies between domain concepts and DSL concepts implicit. The implicit dependencies are easily broken or violated when the system evolves.

The mapping between the different DSL concepts to components and program code is tangled and coupled. Again this is due to implicit dependencies within the implementation of the compiler. Therefore currently changing a DSL requires often reconsidering the whole implementation.

Because of the changes in the DSL compilers, the generated programs each DSL compiler produces change as well. Consequently the composition of the generated programs must be able to cope with these changed programs. Therefore research is also conducted on the composition of those generated programs. This composition to achieve an integration of their respectively generated programs is quite hard and is often a manual process. The integration of generated programs often involves multiple modifications to a generated program at different locations. These required changes should happen obliviously and should be propagated accordingly. Furthermore the resulting integration might cause particular undesired interferences that break the functionality of the generated program.

3.3. *Countering the Evolution Difficulties*

The dependencies between the domain concepts and their implementations, must be made explicit. This way the changes to a part of the system can be traced back to the dependent parts, allowing us to estimate and examine the impact on the dependent parts. In order to make the dependencies more explicit, the implicit dependencies are extracted out the parts of the system and new mechanisms have been conceived to establish these dependencies. This way, the modularity and evolvability of the overall system is thus increased.

We focus on the evolution of DSLs on two levels: evolution of the DSL concepts and evolution of the DSL implementation. We conceived three mechanisms to make the dependencies more explicit. The first one handles the dependencies of the domain knowledge and the DSL concepts to facilitate the evolution of the domain and the DSL. The other two are complementary mechanisms to facilitate the evolution of the DSL implementation. The second one focuses on the dependencies within the implementation of a compiler for a specific DSL and the third one focuses on the dependencies that arise when composing the different generators together.

The first mechanism follows a concept-centric approach [5, 6] that manages and tackles the evolution of the DSL concepts. This approach bridges the remaining “gap” between domain concepts and DSL concepts, making it possible to trace the changes at the domain level to changes of their corresponding DSL concepts. The domain knowledge provides a basis for reasoning about the impact of a change in the DSL language on the domain level itself but also on the DSL level. As a result, insuring consistency in the concepts used in the language becomes a lot easier.

The second mechanism is a new DSL development technique called the Linglet Transformation System (LTS [4]). A DSL implementation consists of stand-alone, modular and reusable language modules. In contrast to other transformation systems, the language modules of LTS are composed via an explicit composition mechanism in a language specification. The composition mechanism takes of care of the necessary inputs that are required by a module and handles the results produced by a module. The dependencies of the modules are made explicit and are external to the modules, hereby reducing the complexities involved during the evolution of a DSL.

The third mechanism is aimed to increase the composability of individual generators that each define their proper DSL[2]. In general, generators are not designed nor implemented to be composed. We have developed composable program generators using an extension to the technique of logic metaprogramming. These generators identify *integration locations* in their generated program where other program parts can be inserted. The integration itself can be specified in a separate specification. Internal to each generator, the transformations are expressed using logic rules. These rules are written such that a generator can produce multiple implementations for a single program. Possible interferences at the foreseen integration locations can be circumvented by 'laws' which choose another implementation for the generated program.

4. Conclusion

Developing iMedia imposes extreme requirements that cannot be met by traditional software development techniques. A new approach called the iMedia software generation system (IMSGS) based on generative programming, transformation systems and an explicit representation of domain knowledge.

Central to the approach are domain-specific languages, which proved to be a very suitable technology for this kind of development. An iMedia implementation is now the product of the combination of the program parts produced by the compilers of the different DSLs describing an iMedia product.

However the evolution of the overall implementation of our approach was hampered by a series of implicit dependencies. To make these more explicit, we followed a concept-centric approach in DSL design, conceived an implementation mechanism (LTS) to handle the internal dependencies in the implementation of the compilers, and conceived a composition mechanism based on logic meta programming to handle the dependencies in the combination of the different parts of a iMedia implementation. Consequently the consistency of the evolution of the language could be more effectively guaranteed and the modularity of the implementation has been increased. These mechanisms improved the evolvability of IMSGS to the extent that the underlying technology now becomes a feasible option for coping with the extremities of iMedia software development.

References

1. K. Beck. Extreme Programming Explained - Embrace Change. Addison-Wesley, 2000.
2. J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages using logic metaprogramming. Proceedings of GPCE Conference, LNCS 2487, pages 110–127. Springer-Verlag, 2002.
3. J. Clark. Xsl transformations (xslt) version 1.0 w3c recommendation 16 november 1999, 1999.
4. T. Cleenewerck. Component-based DSL Development. In Proceedings of GPCE03 Conference, LNCS 2830, pages 245–264. Springer-Verlag, 2003.
5. D. Deridder. A concept-oriented approach to support software maintenance and reuse activities. In Knowledge-based Software Engineering, Frontiers in Artificial Intelligence and Applications, Vol. 80. IOS Press, 2002.
6. D. Deridder. A concept-centric approach to software evolution - enabling open adaptive software development. OOPSLA 2004 Workshopreader, Workshop on Ontologies as Software Engineering Artefacts, 2004.
7. J. A. Highsmith III. Adaptive Software Development - A Collaborative Approach to Managing Complex Systems. Dorset House Publishing, 2000.
8. M. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions, 2001.
9. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. SIGPLAN Notices, 35(6):26–36, 2000.
10. E. Visser. Stratego: A language for program transformation based on rewriting strategies. LNCS, 2051:357, 2001.