ON THE EVOLUTION OF INTERACTIVE MEDIA IMPLEMENTATIONS [*]

T. CLEENEWERCK AND D. DERIDDER AND J. BRICHAU AND T. D'HONDT
Vrije Universiteit Brussel, PROG
Pleinlaan 2
Brussels, Belgium
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

In today's highly competitive market, media production companies are under continuous strain to maintain a high degree of consumer satisfaction. This is achieved amongst others by providing new kinds of services such as the publication of real-time news feeds over multiple channels (e.g. internet, mobile phone, PDA). It is clear that these new media products pose a number of technical challenges for the construction of the supporting software systems. These challenges can in part be overcome by using advanced content management systems, for instance (e.g. [9], [10]). Besides relieving the media producer of coping with the complex technological side, these systems also provide a way to manage the storage and publication of the vast flow of content.

More recently, media broadcasters have observed that media consumers are no longer satisfied with the traditional situation in which the consumer has to "sit back, relax and consume the media". Hence they need to augment their media-offer with different kinds of interactivity such as online gaming, virtual community building, active TV show participation at home,….

[*] This research is partially performed in the context of the e-VRT Advanced Media project (funded by the Flemish Government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.

The production of this interactive form of media (iMedia) encompasses the publication of a behavioral component, along with the media's traditional content component. This behavioral component specifies the interactivity with the media consumer and is inevitably implemented as a software program. Because of the different nature of this behavioral component and its inherent development-intricateness, there exists an urgent need for a kind of *behavior management system* (BMS). This need is easily motivated when looking at the specific characteristics of the context in which iMedia software development takes place.

First of all, iMedia software development takes place in an environment in which *extreme strict deadlines* constrain the development process. As broadcasting occurs in real-time, missing the broadcast deadline consequently renders the iMedia software completely useless. Unfortunately in traditional software development practices, it is quite common to miss deadlines.

A second characteristic of iMedia is the *extreme short time-to-market* situation. The general media situation is one in which many events and decisions occur in real-time and on extremely short notice. 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 challenge. Imagine having to file a change request, (followed by the traditional analysis-design-implementation triplet) asking for a late change when the system must go live in one hour.

Another characteristic is the current need for *extreme deployment* (i.e. multi-platform and multi-channel media-publication). Since each device and platform has its own set of specific capabilities, such as available display size and memory, this need rapidly forms a bottleneck in the production environment. It is clear that this requirement is difficult to fulfill within the strict deadline / short time-to-market context. Hence it requires appropriate support from the software development approach.

It is within this context of extreme characteristics that we propose an alternative approach to iMedia software development. The approach we propose in Section 2.1 combines existing research from the areas of generative programming, transformation systems and domain engineering. This results in a system that is best described as an iMedia Software Generation System (IMSGS). 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. Such 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 a particular instance is done by the media producer (the domain expert). This stands in shrill contrast to the traditional situation in which the adaptation of the software can only be done by a software programmer.

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 DSL and generative programming technology. So in order to increase the practical feasibility of an IMSGS approach we have investigated how we could make the evolution of the generation system easier to do. We have dedicated Section 3 to this contribution.

2. An iMedia Software Generation System

In most software development approaches, the role of a domain expert is limited to the first phases of the process. During those phases, domain experts inject their expertise and state their requirements for the software to be built. Usually from that point on, the rest of the development process is in the hands of the software expert. This unbalanced participation in the process by both parties often prohibits the domain experts to request last minute changes, and often results in uncontrollable and delayed development times.

In order to meet the extreme characteristics discussed in the introduction, the media producer should get a more active and more prominent role in the iMedia development process. This necessitates advanced software development techniques accompanied by an appropriate software development process. A good candidate for the latter are agile software development processes (e.g. eXtreme Programming [1], and Adaptive Software Development [8]). In our work we focus on providing tool-support for developing the behavioral component of an iMedia product.

2.1. *Domain Specific Languages*

To provide media producers with a more active role in the development of the iMedia software itself, our approach aims at bridging the gap between the media domain expert and the software developer. Central to this approach is the use and development of *domain-specific languages* (DSLs) [12]. 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 hide the DSL programmer from non-iMedia-specific technical issues. Consequently, a well-designed DSL allows a domain expert to write programs in this DSL. 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 and, as a result, extreme deadlines and extreme adaptability can be brought in reach of the media developer.

Clearly, to achieve such an approach, software developers are needed 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. Quite often, these concepts and relations are implicitly present in the implementation of the DSL compiler. In our approach we make these concepts explicitly available in a Domain Ontology. It is the difficult task of the language designer to incorporate the more common language features in the domain concepts and to select the appropriate domain concepts to be represented explicitly in the DSL language

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 ([11], [13], [4]). Transformation systems offer flexible programming languages oriented towards transformation processes. 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. In Figure 1 we sketch the different elements in an IMSGS.

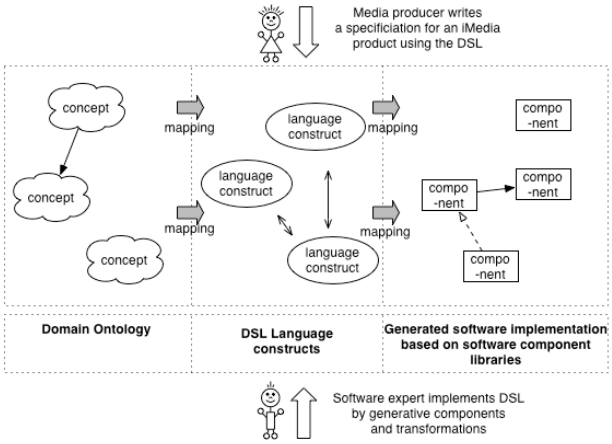

Figure 1 : Overview of the IMSGS.

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, …. 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 program.

2.3. *Illustration*

In what follows we will briefly illustrate the IMSGS approach with a number of examples extracted from a case-study conducted in the context of the e-VRT Advanced Media Project (http://www.xmt.be). The iMedia product domain that is used throughout the examples are TV Quizzes.

To support the media producer in developing his particular iMedia Quiz-software he is provided with a suitable IMSGS. This Quiz-IMSGS is written by the IT-department and consists of a quiz-DSL, and a compiler to transform the quiz-DSL description into an executable software program. As described in the previous sections, the compiler itself contains a transformation system and an intelligent composition engine. Note that the only IMGMS -elements that are visible to the media producer are the language in which he is able to describe his quiz, and the corresponding generated software.

The quiz-DSLs contains several domain concepts (e.g. “round”, “question”, “answer”, “points”), but also comprises concepts necessary for describing execution-oriented concepts (e.g. “control flow”, “random order”, “sequential order”).

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 specialised UI-DSL). Consider the following example description in the quiz-DSL.

```
round R1 {
   question Q1 {
      title :  "Capital city of Belgium"
      type : multivalued
       choices { A:"Paris", B: "Brussel", C:"None" }
      expected answer : B }
   question Q2 {...}
   question Q3 {...}
}

flow R1 {
    questions : 3, random order
   points {
      correct : +2}
}
```

The first part of the description specifies the contents of the quiz-round labelled “R1”. This round contains a set of questions “Q1”, “Q2”, …. The description of the first question indicates that it is a multiple-choice (“multivalued”) question that asks to give the capital city of Belgium. The different answer-choices as well as the correct response are specified. The second

part of the description focuses on the flow of round "R1". There are 3 questions that are asked in a random order. Whenever a question is answered correctly the player gains 2 points. In all the other cases the points remain unchanged.

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 by replacing one line in the flow ("questions: 3, random order" becomes "questions: 3, sequential order (Q1 Q3 Q2)"). 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.

Of course this was the case because the people that did the domain analysis for the Quiz-IMSGS anticipated this kind of change. Hence they had already created the necessary DSL language constructs (random order, sequential order), as well as the necessary generators responsible for generating the corresponding code. It is clear however that a certain point in time, the producer's needs will no longer be satisfied by the capacities of the Quiz-IMSGS. This is not necessarily a result of a lack of expressiveness of the DSL. It could also be that the generators or framework components require updating. We will focus on how we approach the evolution of such an IMSGS in the next section.

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). As we have argued in the former sections we believe that DSL-technology is a valid candidate for improving the way in which iMedia software products are developed. Yet 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, as we will see, 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 (and relationships between these concepts) such as "time", "players". We will also have to update existing concepts such as "flow", "point system", 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. The adjustments to the example are shown below; the specification of the round is omitted because this section remains the same.

First there are a couple of new language constructs needed: a construct "player {...}" 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 (the symbol "&" denotes the and relationship). 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. When the players answer sequentially and the point system has actions for the second player, then the same question may be answered twice. Adding multiplayer functionality involved thus a major change in the language semantics.

```
player P1 {
   points : 0
}
round R1 {
   ...
}
flow R1 {
    questions : 3, random
    players : P1, P2
    turn : all simultaneous
    points {
        first & correct : +2
    }
}
```

Finally the DSL compiler and the underlying component system must be adjusted, refactored and tested. Naturally the additions must not corrupt the existing implementation to avoid unexpected behavioral 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: the mapping from domain concepts to DSL concepts and from 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 coupling and tangling is due 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 Dependencies*

In many software engineering disciplines evolution of the systems is increased by taking care of the implicit dependencies between the parts of the system. Since the implementation approach we propose for iMedia is still a software engineering discipline, the same strategy to increase the evolvability can be used. The dependencies in the IMSGS which are dependencies with the domain concepts and the dependencies of 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.

In our research 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. Additionally the two techniques are developed for the two different kinds of generative programming techniques: transformational and compositional techniques.

The first mechanism follows a concept-centric approach ([6], [7]) 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) [5]. 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], [3]). 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 integration 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. 1.K. Beck. Extreme Programming Explained - Embrace Change. Addison-Wesley, 2000.
2. J. Brichau, K. Mens, and K. De Volder. Building composable aspectspecific languages using logic metaprogramming. In D. Batory, C. Consel, and W. Taha, editors, Proceedings of GPCE Conference, volume 2487 of LNCS, pages 110–127. Springer-Verlag, 2002.
3. J. Brichau. Declarative Metaprogramming for a Language Extensibility Mechanism. In Workshopreader ECOOP 2004, Workshop on Reflection and Meta-Level Architectures, 2004.
4. J. Clark. Xsl transformations (xslt) version 1.0 w3c recommendation 16 november 1999, 1999.
5. 4.T. Cleenewerck. Component-based DSL Development. In Proceedings of GPCE03 Conference, Lecture Notes in Computer Science 2830, pages 245–264. Springer-Verlag, 2003.
6. 5.D. Deridder. A concept-oriented approach to support software maintenance and reuse activities. In T. W. et al., editor, Knowledge-based Software Engineering, Frontiers in Artificial Intelligence and Applications, Vol. 80. IOS Press, 2002.
7. 6.D. Deridder. A concept-centric approach to software evolution - enabling open adaptive software development. Technical report, To appear in OOPSLA 2004 Workshopreader, Workshop on Ontologies as Software Engineering Artefacts, 2004.
8. J. A. Highsmith III. Adaptive Software Development - A Collaborative Approach to Managing Complex Systems. Dorset House Publishing, 2000.
9. Mindhouse. Mindset : Interactivity and Commerce through the Moving Image.
10. Sublime. Sublime iTV Suite: The Professional's Choice for WYSIWIG Creation and Editing of MHP iTV Applications
11. M. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions, 2001.
12. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. SIGPLAN Notices, 35(6):26–36, 2000.

13. E. Visser. Stratego: A language for program transformation based on rewriting strategies. Lecture Notes in Computer Science, 2051:357, 2001.