SharpLudus revisited: from ad hoc and monolithic digital game DSLs to effectively customized DSM approaches

Andre W. B. Furtado, Andre L. M. Santos, Geber L. Ramalho
Center of Informatics - Federal University of Pernambuco
Av. Professor Luís Freire, s/n, Cidade Universitária,
CEP 50740-540, Recife/PE/Brazil
+55 (81) 2126-8430
{awbf, alms, glr}@cin.ufpe.br

ABSTRACT
This paper describes our experience in improving an ad hoc approach for creating domain-specific languages targeted at digital games, replacing it by a customization of more structured approaches in the domain-specific modeling literature. We give special focus on the benefits of partitioning the target game domain into prioritized sub-domains, as well as on promoting game engines to domain frameworks that can be more seamlessly consumed by generated code. A case study for the arcade games domain is also presented for illustration and evaluation purposes.

Categories and Subject Descriptors
D.1.7 [**Programming Techniques**]: Visual Programming.

D.2.2 [**Software Engineering**]: Design Tools and Techniques – Computer-aided software engineering (CASE), Software libraries.

General Terms
Design, Standardization, Languages.

Keywords
Digital games, domain-specific languages (DSLs), visual modeling, software product lines (SPLs), software factories.

1. INTRODUCTION
Similarly to many other domains in software development, digital games development is a peculiar domain to which software reuse processes and techniques, including software product lines (SPLs), domain-specific modeling (DSM), domain-specific languages (DSLs) and generators cannot be applied as is [1]. The challenges of such a domain include the specific engineering complexities of game development [2], the traditional ad hoc, low-level development techniques historically employed in the domain [3], the highly interlaced integration of game components [4] and other specific hindering factors for the domain engineering and automation of the digital games domain [1].

Back in 2006, we presented one of our first experiments towards overcoming such challenges [5]. We explored the state-of-the-art in game development (multimedia APIs, visual game creation or "click-n-play" tools, and game engines) as well as how DSLs and code generators could be built atop game engines to provide increased layers of abstraction. Such a discussion is not to be repeated in this paper. We then introduced SharpLudus, an ad hoc "game factory" aimed at automating the generation of adventure games by the use of a DSL called SharpLudus Game Modeling Language (SLGML). SharpLudus and SLGML revealed valuable insights in the application of DSM to game development. Although they were not aimed at describing a more comprehensive process for performing domain-specific game development, they worked as a "spike solution[1]" towards one.

In this paper, we share the outcomes of evolving the ad hoc SharpLudus approach into a more refined DSM process, entitled *Domain-Specific Game Development* [6], which is a customization of state-of-the-art DSM approaches in the area by Greenfield & Short [7], Kelly & Tolvanen [8], and Almeida [9]. We do not attempt to present a new approach to DSM, but focus on revisiting the original SharpLudus work in the light of the customized approach. We evaluate SharpLudus in retrospection to identify which of the original guidelines are still valid as proposed and which were improved in order to increase their efficiency and/or applicability.

The remainder of this paper is organized as follows. Section 2 provides context on the Domain-Specific Game Development approach, so that Section 3 can then contrast it with the original SharpLudus project. Section 4 explores how we conceived the languages of the ArcadEx game SPL, which instantiates the approach to the arcade games domain. Finally, Section 5 briefly concludes about the presented work.

2. DOMAIN-SPECIFIC GAME DEVELOPMENT
Our customized approach for developing game SPLs, entitled Domain-Specific Game Development (DSGD) [6], is an iterative process ultimately focused on the creation of core domain assets, such as DSLs and generators, to automate prioritized sub-domains of digital games. In a combination of top-down and bottom-up approaches, we call our process "edge-center", in which the problem domain (vision, scope, features, sub-domains prioritization, etc.) is elaborated in alternations with the solution domain (source code, reference architectures, components, etc.). Our main paper about the approach structure [6] provides

[1] According to eXtreme Programming (XP), a spike solution is a simplified program created to explore longer-term potential solutions. It typically addresses the problem under examination and ignores other concerns, being "not good enough to keep" but key to carry on experimentations and reduce risks.

additional information as well as a discussion on related work in the digital game and DSM areas, also not to be repeated here.

As illustrated in Figure 1, by having tasks in the problem domain to complement tasks in the solution domain, each spiral cycle culminates with the creation, or enhancement, of DSLs, generators and eventually other core domain assets for a specific game sub-domain. In a subsequent iteration, another sub-domain of the target game SPL domain is prioritized and elaborated through the same problem and solution domain tasks (the spiral) suggested by the approach.

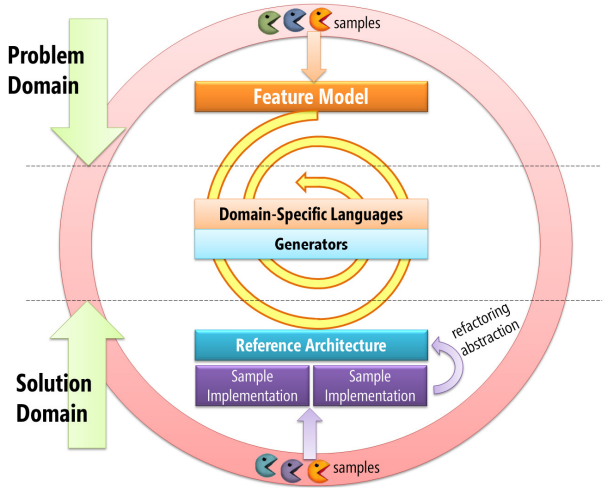

Figure 1. The Domain-Specific Game Development *edge-center* approach [6].

DSGD groups its tasks in four logical phases, refined during each iteration. The *Game Domain Envisioning* phase is focused on establishing a high-level overview and common understanding of the domain to be approached. It avoids the ambiguous and blurry universe of game genres by suggesting that a game SPL domain should not be solely described by means of a genre name (such as "racing games", "card games" or "fighting games"), but by a description of what is expected for **core game dimensions**, such as *player* (number of players, co-playing modes, score system, etc.), *flow* (levels, screens, rooms, etc.) and *entities* (types, main characters vs. non-playable characters breakdown, properties, etc.). Custom core game dimensions, such as the *battle system* for a role-playing game (RPG) domain, are encouraged.

The *Game Domain Analysis* phase provides guidelines on which game samples to evaluate and how to evaluate them, ending up with a set of feature models [10] for the ongoing iteration. It builds or improves the domain vocabulary and partitions the game SPL domain into more specific sub-domains, for which more effective DSLs and generators can be created. Finally, it assesses the game domain automation potential by identifying sub-domain candidates for automation and prioritizing them.

The guidelines suggested by DSGD to identify game sub-domains as automation candidates include **using the natural categorization of the domain** [11] by scanning the feature model that resulted from extracting the domain commonality/variability. Closely related features are normally good candidates to pertain to a same sub-domain, while isolated features also provide good hints [12]. **Relying on the knowledge of the domain expert** [13] helps to further break down the characteristics of the game samples. **Considering core game dimensions and features** directly derived and elicited from them as sub-domain candidates is also recommended, as well as **investigating how sample implementations and game engines modularize types** (classes, interfaces, enumerations, etc.). For example, game engine modules and sub-modules can provide hints on possible sub-domain candidates. A final guideline to identify sub-domains for automation is to **investigate repetition in sample implementations** [12]. If some piece of design or code repeatedly appears in a sample or across samples, even if the repetition instances are not exactly the same, it is likely that a machine can do some parameterized copying and pasting, and it is worth to try to find a sub-domain there.

In order to prioritize the sub-domain candidates, DSGD suggests practitioners to consider: **previous automation evidence** (existing modeling languages and modeling/generation tools for the sub-domain); **integration** (easiness of plugging in new or already existing modeling languages and tools into the game product line); **coverage** (whether the sub-domain covers a bigger amount of features when compared to other sub-domains, or more important features than other sub-domains considering the features were prioritized somehow [9]); **development productivity** (how much effort will developers save if the sub-domain is automated? This can be measured by the expected size of the artifact the sub-domain automation is supposed to generate, in average, for the developed games); and **development abstraction** (if implemented manually, how complex or error-prone are the artifacts supposed to be generated by the sub-domain automation? Examples of artifacts with a high error-prone and complexity rates are code or configuration files that deal with too many literal values and constants or require lots of repetition but yet a few customizations that could be missed, code that requires application of design patterns and code that ensures a non-functional concern such as performance or security is satisfied).

The *Application Assets Creation* phase customizes the typical Domain Design and Domain Implementation activities of Domain Engineering [14] to create application core assets for the target digital games domain. Mirroring the findings of Kelly & Tolvanen [8] on component libraries, such activities build or improve a domain-specific game architecture by promoting game engines to domain frameworks [15]. In other words, they ensure that game engines support the domain variability, framework completion (i.e., they can be seamlessly consumed by generated code) and extensibility so that the generated games are not constrained to the built-in behaviors supported by the game SPL. Likewise, this phase also guides the integration of reusable game components (such as an on-screen keyboard or a heads-up display) into the domain-specific game architecture.

Finally, the *Development Assets Creation* phase enables the design and implementation of DSLs, generators, automated guidance and other development core assets, integrating them to the game SPL. They characterize the variability level for each prioritized sub-domain (ranging from routine configuration to creative construction [16]), design or improve DSLs for it (including their abstract and concrete syntaxes as well as cross-language integration), design and implement generators and the development environment (IDE) integration.

3. SHARPLUDUS REVISITED

Thinking in retrospect, many lessons were learned in the journey from the original SharpLudus project [5] to the current Domain-Specific Game Development approach [6]. Describing those is the main goal of this subsection.

From one hand, the original assumption that DSLs are underexplored in the context of the digital games development domain still holds true. In the lack of better metrics, case studies developed with SharpLudus reported the generation of dozens of classes and thousands of source lines of code in a couple of hours of development. The DSGD approach results are also encouraging, reporting a four to five times faster development than approaches employing only game engines. Likewise, although click-n-play tools still occupy their niche by enabling non-programmers to quickly assemble game functionalities via a drag-and-drop user experience, visual development and modeling environments targeted at actual game programmers and the game industry in general are still not mainstream. We can find some level of domain-specific development in game engines, because they evolved from APIs into a more comprehensive toolset encompassing script languages, such as UnrealScript. However, such languages are still at a fairly low programming level, raising concerns as to the level of abstraction they offer [17].

On the other hand, the original SharpLudus project and the current DSGD approach diverge in some relevant points. Obviously, the main difference is that while SharpLudus was an ad hoc instance of a game SPL, DSGD defines guidelines for the creation of game SPLs, being a more comprehensive and mature approach whose foundations were tightly built from Domain Engineering [14] and other software reuse and abstraction concepts, such as SPLs and DSM [7, 8, 9].

Already concerned with the problems entailed by the ambiguity of game genres, SharpLudus suggested that the target game domain had to be described by means of a "product line definition". DSGD evolved the concept by requiring the game SPL to be described by means of the expectations implied by *core game dimensions*, which are not overly generic or specific to a game domain. Nonetheless, a more important difference is how such assets are used once created. While the product line definition (together with the domain vocabulary) was used by SharpLudus as a direct input to DSL design, in DSGD the core game dimension expectations, together with other assets such as the identified non-emotional requirements for the domain, end up as input for the creation of feature models [10]. Such intermediate step makes the identification of the commonality and variability of the domain much more evident, and is key to identify and prioritize sub-domains, leading to more expressive and effective DSLs and generators. In fact, SharpLudus lacked a more structured Domain Analysis phase in which guidelines are provided for the selection and analysis of domain samples, as suggested by Greenfield & Short [7] and Almeida [9].

Probably one of the most notable evolutions from SharpLudus to DSGD is the sub-domain breakdown employed by the latter. Although the SharpLudus Game Modeling Language (SLGML) is a *domain-specific* language, thinking in retrospection we concluded it is not atomic enough. It encompasses too many concepts that, despite of being related, could have been explored by SPLs in a much more effective way if separated. In short, SharpLudus lacks more specific, atomic yet integrated languages.

For instance, SLGML encompasses the concepts of audio, entities, events and game flow altogether. Having all of them in the same modeling diagram would provide a confusing user experience. As a result, the core of SLGML's concrete syntax focused on only one of such concepts: game flow (Figure 2). In other words, the modeling part covered only a subset of the application. The management of other concepts such as entities and events was performed in normal lists and dialogs (Figure 3), launched as custom property editors assigned to the properties of the domain's root concept (the "adventure game").

Figure 2. SLGML: modeling focus was on the game flow.

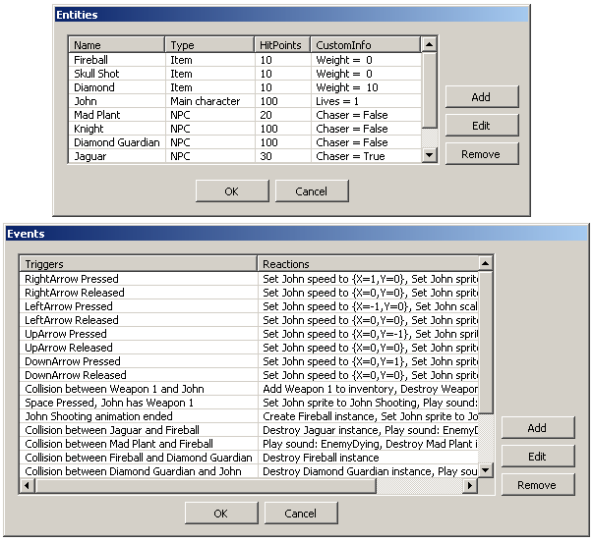

Figure 3. SLGML: concepts such as Entities and Events were managed through lists and dialogs

We believe this approach is not optimal for at least three reasons. First, custom dialogs and lists built from standard UI controls do not typically provide the desired level of abstraction for a specific domain. In such an approach, the concrete syntax of the domain concepts gets mixed with concepts of the user interface API domain, such as buttons and list boxes. Likewise, in such an approach game developers and designers are more likely to deal with instances of the concepts in isolation, as Figure 3 shows: the entities of the game are described one by one in the list box, but interesting relationships between them, such as whether anything happens if they collide, are not described.

Second, although the aforementioned problems can be mitigated by the creation of custom, refined UI controls, such a workaround may require a considerable amount of effort. In fact, the excessive creation of custom UI controls for modeling purposes

seems to be a duplicated effort, considering that this is the same role of language workbenches (toolset through which DSLs and generators can be effectively implemented).

Finally, having all instances of a concept to be described in the same list may decrease the overall cohesion of the models. This was observed at least in two opportunities in SharpLudus. All animations of a game were defined together in the same list, but each set of animations was used only by a specific game entity, i.e., the sets had no relation to each other. Hence, it would make more sense to have them managed from their respective entity instead of together. Likewise, game events were defined in the same list, but their triggers actually came from multiple sources (entity collision, timers, player input, etc.). Hence, the modeling experience could be improved and made more cohesive if each event was managed from the source concept that triggers it.

Learning from this experience, DSGD advocates that the target game SPL domain should be broken down into atomic sub-domains. Examples of such sub-domains are the transition between game scenes or screens, entity or screen timers responsible for triggering events, the collision relationship between game entities and the possible graphical representations of heads-up displays. Such sub-domains are too atomic to comprehensively define a game by themselves. On the contrary, different features of a game fall under such sub-domains. The game is the sum of the features distributed in the sub-domains.

As the target game SPL domain is broken down in sub-domains, DSGD's edge-center spiral focuses on approaching one prioritized sub-domain at a time. The sub-domain chosen for a given iteration has its feature model detailed, is mapped against existing source code from samples, has corresponding modules implemented in the domain-specific game architecture to support its commonality and variability, and ultimately leads to the creation of very specific, atomic DSLs and generators.

DSGD provides guidance on how to characterize the variability for a sub-domain, which will determine the concrete syntax of its DSLs. The simplest form of variability is *routine configuration*, in which simpler, tree-like DSLs, such as wizards or feature-based configuration, are used to select a subset of features when configuring a product. On the other end of the variability spectrum, *creative construction* requires complex, graph-like DSLs, such as programs and models to be created using textual or visual languages [16]. Similarly, techniques for developing transformations are more detailed in DSGD, which elaborates how template-based code generators can be achieved by migrating source code from the reference implementation to templates, annotating it with tags and scriptlets that bind the code to the DSL.

As a result of the sub-domain breakdown, some benefits can be observed. First, it outputs **more expressive DSLs and generators,** since each is responsible for a well-defined subset of the target SPL domain. The most important entities of the target SPL domain end up being represented as first-class concepts in DSLs, instead of lower-level abstractions based on lists and UI controls. Second, it allows an **incremental delivery of value.** DSGD is not an all-or-nothing automation approach. Even if the first version of a game SPL automates only one sub-domain, delivering a single DSL and generator, game designers and developers can already start harvesting the benefits from it. Future versions of the SPL can deliver new assets or improve already existing ones, automating more sub-domains incrementally. Finally, each sub-domain is evaluated for its automation potential, providing **more confidence to ensure the sub-domains with the best return on investment are the ones prioritized for automation.**

On the other hand, the sub-domain breakdown may require cross-DSL integration, which has a lot of challenges on its own. For instance, while it is quite straightforward to consume one class from another in the source code level, it is not similarly simple to make one language access the concepts of another language, as well as ensuring their references are always in sync and updated. While cross-DSL integration is not a topic approached by SharpLudus, DSGD provides some guidance on how that can be achieved, exploring the concepts of name-based references, model bridges and model buses.

Some SharpLudus contributions to the *development core assets* area still apply to DSGD, such as the creation of semantic validators so that DSL users can catch modeling errors at design time. Another contribution that was kept is the guidance for game SPL designers to choose a language workbench.

In the *application core assets* area, both SharpLudus and DSGD advocated for using game engines, a state-of-the-art resource in game development, as a central piece of the domain-specific game architecture. However, DSGD brings an important improvement: promoting the game engine(s) to a *domain framework* which can be seamlessly consumed by generated code and therefore is able to move complexity away from code generators [15]. This turns out to be extremely important since code generators are more difficult to maintain than a framework.

SharpLudus already had concerns related to making the generated games flexible and extensible enough so that the built-in SPL features could be complemented with custom, developer-added features as a result of creative processes in the domain. It suggested the double-derived design pattern [15] to be employed, in which instead of a single class, a pattern of two classes is generated for a given domain concept. The base class contains all of the generated method definitions as virtual functions; the derived class contains no method definitions but is the one that is instantiated, which allows the user, in a partial class or similar language technique, to override any of the generated functions with their own version. DSGD builds on top of this discussion by suggesting more variability techniques and extensibility channels to the domain framework, such as: aggregation/delegation, inheritance, parameterization, overloading, properties, dynamic class loading, static libraries, dynamic link libraries, conditional compilation, frames, reflection, aspect-oriented programming and design patterns [18].

4. A CASE STUDY FOR ARCADE GAMES

In order to evaluate DSGD, we instantiated it to a couple of game domains, such as isometric adventure games, RPG games and mobile touch-based games. The SPL we created for 2D arcade games, called ArcadEx, is the most interesting from an evaluation perspective, since it is the one in which the approach was most comprehensively employed.

ArcadEx's expectations for the core game dimensions eliminated any ambiguity and blurriness caused by the "arcade" game genre.

They define ArcadEx as a SPL focused on generating single or multiplayer 2-dimensional arcade games for PCs, with short levels composed by screens containing entities and walls, quick play action (in contrast to more in-depth gameplay or stronger storylines), simple, easy to grasp controllers, iconic characters and eventually rapidly increasing difficulty. Players control main characters and their projectiles that collide with other entities such as non-player characters (NPC) or items. Victory condition is specified by the game designer as (a set of) game events: enemies are defeated, an object is collected, etc.

Iterations were used for the breakdown and prioritization of sub-domains. For each sub-domain, we analyzed samples, extracted and detailed features (resulting in a feature model of more than 150 features), inspected and implemented code, and refined a domain-specific game architecture. We ultimately came up with four DSLs (Figure 4) and code generators, instead of a single bloated DSL that lacks conciseness and maintainability.

In the first iteration, we approached the **screen transition** sub-domain, including the different triggers that make an ArcadEx game to move from one screen to the other, such as an input action or a timer. Following the extensibility guidelines, we added support for custom transition events, to be programmed by game developers but could still be referenced from the models. This resulted in a first version of the *ScreenTransitionDSL*, which was then renamed to *GameDefinitionDSL* in the second iteration after we concluded that such a DSL was the one through which developers could also specify the **top-level properties of a game** (such as its window mode, resolution, etc.). In the third iteration, we refined the *GameDefinitionDSL* as a result of approaching the **screen background** sub-domain, allowing game developers and designers to assign static background pictures to screens instead of manually programming code to render them. In the fourth iteration, we explored the **background music** sub-domain. Properties such as what music asset to play as background music were added to each screen, along with the background music behavior such as "start new music", "keep playing the current one", etc. After these four iterations, game developers and designers had a suitable version of the *GameDefinitionDSL*, which allowed them to perform various screen flow management tasks in a higher level of abstraction, via DSLs and models. Nonetheless, other game features still had to be programmed in the low level.

In order to improve the (manual) testability and expedite experimentations of the SPL, we then approached the **input mapping** sub-domain, i.e., the mapping of the gamepad (controller) buttons to keyboard keys. This allowed games to be played with the keyboard, although input events in the models were specified by means of gamepad buttons. As a result, the *InputMappingDSL* was created. Subsequent iterations explored the **entity definition** sub-domain, including entity states and animations, resulting in the *EntityDSL* through which the "things and beings" of ArcadEx games could be modeled, instead of programmed. Many iterations were required to refine this DSL, approaching domains such as the **declaration of collision interest** between entities, **entity input handlers** (single-button, 8-direction movement, etc.), **entity event reactions** (create entity, destroy entity, switch state, etc.), **entity-based timer events**, and others. Due to the sub-domain prioritization guidelines, some of the *EntityDSL* refinements were alternated with the creation and refinement of another DSL, called *ScreenDSL*. Such a DSL is the result of prioritizing sub-domains related to screen contents, such as **heads-up displays** (textual, icon or progress bar), the **placement of entity instances** in a screen and **screen-based timer events**.

ArcadEx's DSLs were continuously revisited as the game SPL evolved and new sub-domains were approached. For example, the *GameDefinitionDSL* was updated after the **scrolling backgrounds** sub-domain was chosen for automation. Likewise, a few more complex concepts required cross-DSL integration, churning the DSLs. For instance, we enabled textual heads-up displays from the *ScreenDSL* to reference an entity property defined in the *EntityDSL*, such as the number of remaining hit points.

Evidently, the creation and refinement of the DSLs was not the only deliverable of each iteration. Code generators for the DSLs were also being developed and refined. In order to move complexity away from the multiple generators, incremental layers were implemented on top of the FlatRedBall game engine[2], chosen to be the heart of ArcadEx's domain-specific game architecture. With such layers, FlatRedBall was gradually promoted to a domain framework. Any extra plumbing required to consume the engine was eliminated by means of adapters. Special attention was given to offer easy-to-consume extensibility hooks plugged into FlatRedBall for unforeseen game behaviors. In summary, the promoted FlatRedBall game engine encapsulated all of the commonality of the domain, while still supporting the variability expressed by the DSL and extensions.

Case studies involving the creation of games through the ArcadEx SPL brought up some benefits of using the Domain-Specific Game Development guidelines: we got incremental delivery of value via the prioritized sub-domains automation, a reduced complexity to consume game engines (promoted to domain frameworks) from the generated code and domain-specific assets tailored to the unique characteristics of the envisioned family of games. From an end-user perspective, ArcadEx games are developed in one-fifth to one-fourth of the time required to develop them using the game engine alone. Such results are in line with MDD improvements measured for other areas [19].

On the other hand, we initially observed that the automated sub-domains presented reduced levels of flexibility, i.e., the SPL's built-in assets did not comprehensively cover the game domain variability. We overcame such a drawback by providing more extensibility hooks so that unpredicted behaviors could still be programmed by hand and integrated to the models as extensions. Many of these extensions were then incorporated to the game SPL's built-in feature set in later iterations.

Another challenge that we faced with the DSGD approach relates to backward compatibility. More than once, refined versions of the DSLs broke existing model instances. For such scenarios, we suggest that migration tools are developed in order to assist game developers and designers to move their models to the new versions of the DSLs. Finally, we strongly recommend practitioners to move the implementation of helper methods and

[2] flatredball.com

supporting APIs for cross-language integration away from code generators, as they are one of the hardest SPL assets to maintain.

5. CONCLUSIONS

Combined with the promotion of game engines to domain frameworks, the replacement of monolithic game DSLs by a set of more atomic DSLs targeted at specific sub-domains is an interesting move for game SPLs. It breaks down game development tasks into more granular and automatable chunks, which can be prioritized and implemented accordingly. We believe many of our lessons learned can be employed to domains other than game development, although this is out of the scope of this paper.

Although our Domain-Specific Game Development guidelines do not constitute a comprehensive Domain Engineering process per se [14], the benefits of its systematic approach and the results obtained so far make us believe they are a step in the right direction for automating more of the game development domain. As a result, we hope game SPLs are able to free up valuable time that can be allocated in creative, experimentation and prototyping tasks responsible for making each game title unique and distinct.

6. REFERENCES

[1] Furtado, A. W. B.; Santos, A. L. M.; Ramalho, G. L. Streamlining domain analysis for digital games product lines, Proc. of the 14th Int'l. Conf. on Software product lines: going beyond, Springer-Verlag, 2010, pp. 316-330.

[2] Blow, J. Game Development: Harder Than You Think, ACM Queue, vol. 1, no. 10, 2004, pp. 28–37.

[3] Reyno, E.M.; Cubel, G.A.C. Model-Driven Game Development: 2D Platform Game Prototyping, Proc. Game-On 2008, 9th Int'l Conf. Intelligent Games and Simulation, EUROSIS, 2008, pp. 5–7.

[4] Folmer, E. Component Based Game Development: A Solution to Escalating Costs and Expanding Deadlines? Proc. 10th Int'l ACM SIGSOFT Symposium Component-Based Software Engineering, Springer, 2007, pp. 66–73.

[5] Furtado, A. W. B.; Santos, A. L. M. Using Domain-Specific Modeling towards Computer Games Development Industrialization, The 6th OOPSLA Workshop on Domain-Specific Modeling (DSM06), 2006.

[6] Furtado, A. W. B.; Santos, A. L. M.; Ramalho, G. L.; Almeida, E. S. Improving Digital Game Development with Software Product Lines. IEEE Software Magazine, vol. 28, no. 4, Engineering Fun, 2011.

[7] Greenfield, J.; Short, K. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004.

[8] Kelly, S.; Tolvanen, J.-P. Domain-Specific Modeling: Enabling Full Code Generation. Wiley, 2008.

[9] Almeida, E. S. RiDE: The RiSE Process for Domain Engineering, Ph.D. Thesis, Federal University of Pernambuco, 2007.

[10] Kang, K.; Cohe, S.; Hess, J.; Nowak, W.; Peterson, S. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[11] Maiden, N.; Sutcliffe, A. A computational mechanism for parallel problem decomposition during requirements engineering. In: 8th International Workshop on Software Specification and Design, Germany, pp. 159–163 (1996).

[12] Lucrédio, D.; Fortes, R. P. M.; Almeida, E. S.; Meira, S. R. L. Performing domain analysis for model-driven software reuse. In 10th International Conference on Software Reuse, Beijing, China, 2008;

[13] Lee, K.; Kang, K. C.; Lee, J. "Concepts and guidelines of feature modeling for product line software engineering," in 7th Intl Conference on Software Reuse (ICSR), Austin, Texas, 2002, pp. 62–77;

[14] Czarnecki, K.; Eisenecker, U. W. Generative Programming: Methods, Tools, and Applications. Addison Wesley, 2000.

[15] Cook, S.; Jones, G.; Kent, S.; Wills, A. C. Domain-Specific Development with Visual Studio DSL Tools, Addison-Wesley Professional, June 2007.

[16] Czarnecki, K. Overview of generative software development, in Int'l Work-shop on Unconventional Programming Paradigms, France, Sep 15-17, 2004, ser. LNCS, Vol. 3566. Springer, 2005, pp. 326–341.

[17] Dobbe, J. A. Domain-Specific Language for Computer Games, MSc dissertation, Department of Software Technology, Delft University of Technology, 2007.

[18] Anastasopoulos, M.; Gacek, C. Implementing Product Line Variabilities, in Symposium on Software Reusability (SSR), Toronto, Canada, 2001, pp. 109–117.

[19] Kelly, S. Domain-Specific Modeling: MDD that Works, blog, 17 Mar. 2010; http://bit.ly/g1KyWp.

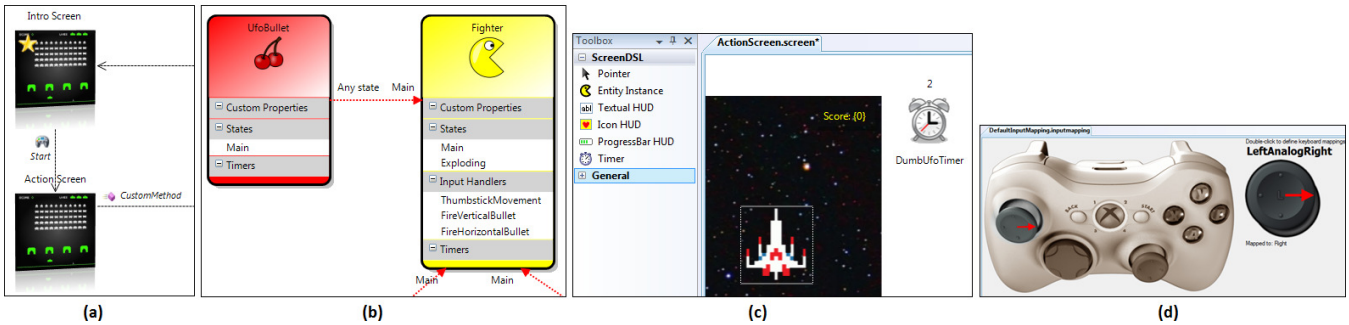

Figure 4. ArcadEx DSLs: (a) GameDefinitionDSL, (b) EntityDSL, (c) ScreenDSL, (d) InputMappingDSL