Semantic Deltas for Live DSL Environments

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Science Park 123, Amsterdam, The Netherlands
storm@cwi.nl

***Abstract*—Domain-specific languages (DSLs) require IDE support, just like ordinary programming languages. This paper introduces *semantic deltas* as a foundation for building *live DSL environments* to bridge the "gulf of evaluation" between DSL code and the running application. Semantic deltas are distinguished from textual or structural deltas in two ways. First, they have meaning in the application domain captured by the DSL. Second, they can be interpreted at runtime so that the behavior of the running system adapts to the evolved DSL code. Semantic deltas have the potential to support back-in-time debugging, application state persistence, version control, retroactive updates and exploring what-if scenarios. I present early experiences in building a live DSL environment and identify areas for future research.**

I. Introduction

The goal of live programming is to bridge the "gulf of evaluation" [11],—the cognitive impedance mismatch between the source code of a program and its dynamic manifestation at runtime. Concretely, live programming aims to provide continuous semantic feedback on the state of running programs. The program can be stopped, restarted, its state can be inspected. Changes to the program are immediately reflected in the running instance.

Programming can be seen as a complex process of turning problem space requirements into solution space artifacts. The original problem specification is often "lost in translation". As a result, evolving software systems is time-consuming and error-prone. An important goal of DSLs is to bridge this "gulf of encoding" by providing high-level notations, tailored to specific problem domains [17]. They can be seen as specification languages to directly express *requirements* from which the actual code is automatically generated. Problem-oriented notations improve communication with stakeholders and domain experts. A *live* DSL environment may amplify this benefit: the DSL code is not only easier to understand and maintain, but can also be inspected, tested and explored during development.

Turning DSL implementations into live DSL environments requires reasoning about changes to DSL code. This is where *semantic deltas* come in. The essential idea is to represent changes to the source code of a DSL program as explicit, semantically meaningful change packages. Such deltas are then interpreted on the runtime state of the running application. Moreover, the user may simultaneously use the running application, for instance through interacting with a GUI. Such application-level events can be represented in the same way. As a result, both kinds of interactions (source code and running application) can be treated and recorded uniformly. The result is that the evolution history and application state history are unified.

Semantic deltas as a foundation for live DSL enviroments have many potential advantages:

- *Liveness* Changes to the program are immediately applied to the running application. As a result, the user of such a system can enjoy immediate feedback on the effect of changes to the program.
- *Persistence* Recording the sequence of deltas and storing it on disk allows a session to be restarted at any later moment in time and facilitates post-mortem analyses.
- *Version control* Storing source-code semantic deltas provides a basis for domain-specific version control.
- *End-user programming* Effacing the distinction between source changes and runtime state changes has the potential to enable end-users to modify their applications. Meta-level and object-level can be modified using the same mechanism.
- *Backtracking* If deltas maintain enough information to be inverted, this enables going back in time. In essence, application-level undo/redo and reverting to a previous version become one and the same thing.
- *Retroactive updates* If dependencies between deltas are known, states (or program versions) in the past can be updated, after which deltas that are still valid are replayed.
- *Time branching* To explore what-if scenarios, an application (program + running instance) can be forked, so that the resulting two instances follow different branches in time.
- *Merging* Given an operation-based merge algorithm, forked applications can be merged again. As with all merging, this might require interactive conflict resolution.

In the rest of this paper I present a simple example of a live DSL environment and discuss related work, open research questions and opportunities to push forward the development of live DSLs.

II. Example: LiveQL

A. A Live DSL Environment

LiveQL is a live version of a simple domain-specific language (DSL) for modeling questionnaires[1]. Figure 1 shows an interactive session with LiveQL, the demo environment

[1]The code for LiveQL is available at https://github.com/cwi-swat/LiveQL.

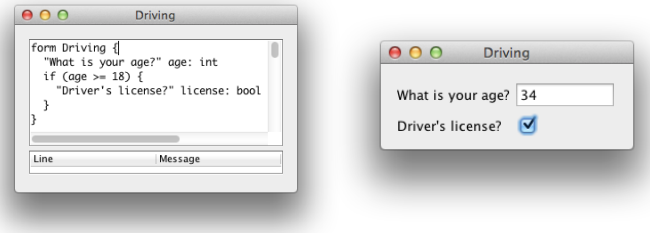

Fig. 1. Screenshot of the LIVEQL demo-environment

TABLE I
CLASSIFICATION OF SEMANTIC DELTAS FOR LIVEQL

AddElse : $IfThen \times Stat \to IfThenElse$
RemoveElse : $IfThenElse \to IfThen$
InsertStat$_i$: $Block \times Stat \to Block$
RemoveStat$_i$: $Block \to Block$
SetCondition : $Conditional \times Expr \to Conditional$
SetExpression : $Computed \times Expr \to Computed$
SetType : $Question \times Type \to Question$
SetLabel : $Question \times String \to Question$
Rename : $Question \times String \to Question$
ToAnswerable : $Computed \to Answerable$
ToComputed : $Answerable \times Expr \to Computed$
SetValue : $State \times Question \times Value \to State$

to develop QL programs. The left-hand side shows a simple editor showing an example QL program.

Questionnaires consist of simple declarations of questions and conditional flow. Each question has a label, a variable name, and a type. There are two kinds of questions: answerable (to be answered by the user, e.g., `age`) or computed. Computed questions have an associated expression that determines its value. Running a QL program results in a graphical user interface with labels and widgets representing questions. Parts of the GUI become (in)visible depending on the value of conditions. An example is the condition `age >= 18` in Figure 1. In turn, conditions may depend on the data entered by users and how computed questions are evaluated.

The QL program in the editor (left of Figure 1) can be edited during the same session. After every edit of the program, the program is parsed and checked for semantic errors, and, if successful, the GUI and runtime state instantly adapt to the new situation.

B. Classifying Semantic Deltas

The signatures of the relevant semantic delta types for QL are shown in Table I. Source-level deltas take the target entity (i.e. AST node) as its first parameter. Additional parameters represent required information to perform the change. For instance, the delta type AddElse is used if an else-branch is added to an if-statement that did not have one before (represented by the IfThen AST type). Such deltas contain the added statement. For blocks – lists of statements enclosed in curly braces – there are InsertStat and RemoveStat deltas, which additionally take an index in the list. Note that there are no deltas defined on expressions, since for QL this level of granularity is not required. The last function is special: this is the delta type representing changes that result from interacting with the running application. The SetValue delta updates the state after the user has entered some data.

TABLE II
A POSSIBLE SCENARIO INVOLVING EDITING A QL DESCRIPTION (Δ_i) AND USING THE GENERATED QUESTIONNAIRE (δ_j) AT THE SAME TIME.

Delta	Semantics	Runtime effect
Δ_1	SetCondition(Q_1, $Q_0 > 0$)	Q_1 becomes invisible
δ_1	Enter 10 in field Q_0	$Q_0 \mapsto 10$, Q_1 becomes visible
δ_2	Enter 2 in field Q_1	$Q_1 \mapsto 2$
Δ_2	ToComputed(Q_2, $Q_0 \times Q_1$)	$Q_2 \mapsto 20$, Q_2 is rendered

An essential characteristic of semantic deltas is that they represent change in terms of the application domain. Each source code delta essentially represents a change in the requirements. At the same time they carry enough information to be automatically applied to obtain a new version of the DSL program and synchronize the running application.

C. Recording Semantic Deltas

Interactions with a LIVEQL editor or questionnaire can be recorded as sequences of semantic deltas. An example of such a user interaction is shown in Table II. Editing the QL program produces semantic deltas Δ_i, – interaction with the generated questionnaire produces semantic deltas δ_i. The end result, however, is a single stream (or change script) modeling the evolution of the QL program and the runtime state of this particular instance. As long as the static semantic constraints of the QL language, and the dynamic constraints of the GUI are not violated, both kinds of changes can be arbitrarily interleaved.

The sequence of deltas can be rewinded to inspect previous states. It can be stored on disk and restarted later. Changes in the past can be retroactively updated, or retracted. It could even be possible to fork a running application into two parallel versions to explore what-if scenarios.

D. Interpreting Deltas

The LIVEQL interpreter renders the GUI corresponding to the structure of the questionnaire and installs the necessary event handlers to respond to user input, and observers to propagate changes to computed expressions and conditions. The current value of questions (as entered by the user or computed by an expression) is stored in a state object (essentially a map from variable names to values). The interpreter thus has the following signature: $run : QL \to GUI \times State$. To make a running QL program adapt to changes to the source, another interpreter is needed, this time with the following signature: $run_\Delta : QL_\Delta \to GUI_\Delta \times State_\Delta$. The function run_Δ is a *delta interpreter*, which maps the domain-specific, semantic deltas to updates to the GUI and the runtime state.

The delta interpreter updates the running application as follows:

- Widgets corresponding to added (or removed) questions are rendered (or removed). Values are converted according to changed types. Expressions of conditions and computed questions are updated. Question labels are updated where needed.
- Changes to the conditional structure of the program are reflected by updating the visibility condition for all current widgets.
- The observer dependency graph is updated to reflect any changes (additions, removals, or substitutions) to expressions in computed questions and conditions.
- All questions that have been marked as dirty in the previous three steps are triggered as being changed to update the rendering on screen. Questions are triggered in the order of occurrence in the QL program.

A SetValue(s, Q, x) delta updates the state so that Q is now bound to x. After that, all observers subscribed to Q are notified of the change.

This initial experiment with LIVEQL is far from finished. However the current state of the prototype shows that there is potential in using semantic deltas for building live DSL environments. Furthermore, the QL language provides a suitable test-case to build more advanced live features based on the foundation of semantic deltas.

III. DISCUSSION

The application of semantic deltas for building live DSL environments raises several important questions. Below I review these questions in the context of related areas of research.

A. What Can We Learn from Version Control?

Semantic deltas can be obtained in two ways: by comparing two consecutive versions of the program, or by letting programmers edit the program by applying the deltas directly. There are many ways to compare two versions of a program: textual (the well-known `diff` algorithm [18]), syntactic [29], and tree-based [1], [2]. Although such algorithms can be used to compare program code, the resulting deltas are generic and not semantic. Consequently, such deltas are hard to interpret in the context of a running application.

So-called "semantic diff" algorithms take the semantics of the compared entities into consideration. Examples are semantic differencing for a procedural programming language [7] and the comparison of object-oriented class models [15], [28]. These algorithms compute deltas in terms of the source language, and consequently cannot be reused outside their domain. Note that, although structural and semantic diffs may be easier to obtain if programs are maintained using a structure editor [3], [20], this is primarily an engineering decision.

The other approach to obtain semantic deltas is to force programmers to edit their programs by directly applying deltas as transformations or "refactorings"[2]. Since semantic deltas are assumed to be complete, it is possible to perform any desired edit this way. The area of evolution-by-transformation could provide inspiration for this model [8], [10]. It is however an open question whether such an approach would scale, and if so, for what kind of languages.

It is well-known that program changes often are not independent. An example in the context of QL would be the deletion of a question. If the corresponding variable is used in an expression, the use of this variable should be removed from the expression. At the source level, consistency of sets of deltas is ensured by static checking of the QL program: no deltas are produced for semantically invalid QL programs. However, applying deltas to the state of the running application might induce more dependencies. Such dependencies have to be respected by the delta interpreter.

Modeling semantic delta dependencies is important for three reasons. First, semantic deltas have to be applied in the right order. If not, the computational effect of a delta might be based on out-of-date data. Second, it is required for *retroactive updates*: going back in time, changing something, and reapplying the deltas after this change. The change in the past, might (transitively) invalidate subsequent deltas that depend on that state. Similar issues emerge when (semi-)automatically *merging* two parallel delta histories.

Independence of changes is studied in the area of software merging [16]. Especially relevant for semantic deltas is operation-based merging [13]: merging two streams of change operations. For instance, the algorithms of [13] require an oracle whether two change operations commute. More recently, similar properties have received attention in the context of the version control system DARCS [24].

B. How Well Do Semantic Deltas Scale?

QL is an arguably simplistic DSL. A question is thus whether the semantic delta approach can be applied to other, bigger, more real-life DSLs. A key characteristic of QL is that all changes can be expressed as deltas by referring to domain entities by name (i.e., questions). For DSLs that do not depend as much on names, different identification mechanisms should be used. In any case the "semantic content" of deltas will be less. How would one, for instance, express *semantic* changes to regular expressions or SQL queries?

Another characteristic of QL is that the resulting system is stateful and interactive. For DSLs that operate in batch-mode the approach could be applied to (possibly interactive) visualizations of the computed end result. An example would be a DSL for syntax definition (e.g., EBNF) where the role of "running system" could be played by an interactive parse tree visualization.

Further research should show whether semantic deltas can be applied to more complex DSLs, such as, for instance, WebDSL [5], – a language for interactive web applications. A first challenge here is that WebDSL implementation is based on code generation. It is completely unclear whether semantic deltas could be used fruitfully without an interpreter. Furthermore, WebDSL's entity modeling language hints at another challenge: how to migrate complex, persistent data?

[2]Note that refactorings are in a sense complementary to semantic deltas: they are assumed to be semantics preserving instead of semantics changing.

Research on coupled evolution might provide concepts and techniques to tackle this problem [6], [9], [22]. Finally, WebDSL – and many other DSLs – are implemented via complex transformation chains involving multiple intermediate representations of the source model. Relating source level semantic deltas to elements of the running system is hard, if not impossible, unless advanced origin tracking techniques are used [21].

Research in omniscient debugging [12], [19] has shown that maintaining execution histories can be made practical. These techniques, however, deal with debugging of arbitrary programs in which the amount of data that has to be stored can be truly staggering. The vision of semantic deltas, however, is *not* to maintain full execution traces, but only the changes that are relevant at the level of the domain. In other words, semantic deltas represent changes that have meaning to *users* of the system. They are much coarser grained than the assignment of a variable. Furthermore, delta-based object persistence frameworks, such as Object Prevalence [26], [27] and EventSourcing [4], have been shown to scale very well.

Finally, it is interesting to investigate how semantic deltas could be used for multi-user systems. In this case, there are multiple streams of semantic deltas that have to be merged in a consistent way. In other words, multi-user applications would require a form of concurrency control. Luckily, there is extensive research in the area of groupware editors, which are also based on concurrent streams of operations that have to be synchronized [25].

C. Towards Live Language Workbenches?

Language workbenches [23] present a significant productivity boost in developing DSLs. Present-day language workbenches provide tools and services for creating integrated development environments (IDEs) for languages. However, to the author's knowledge, none of the currently available language workbenches provide support for developing live DSLs environment. Initial steps towards a fundamentally interpretive style of DSL implementation are being made in the context of the ENSŌ project. A core aspect of ENSŌ is *managed data* [14] which allows users of the system to customize ENSŌ's data definition and manipulation facilities. This could represent a first step towards reusable support for representing semantic deltas, persistence, origin tracking and coupled evolution.

IV. Conclusion

Live programming aims to bridge the cognitive gap between program code and the running system. In this position paper I have presented the concept of semantic deltas as a foundation for live DSL environments. Semantic deltas capture both changes to the source code and changes to application state. This uniform representation may enable powerful live programming features, such as going back and forth in time, retroactive updating, exploring parallel futures, etc. Further experimentation will have to show how well semantic deltas scale, and what tools and techniques from related research areas can be reused.

References

[1] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, June 2005.

[2] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, 1996.

[3] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The MENTOR experience. Technical Report 26, INRIA, July 1980.

[4] Martin Fowler. Event sourcing. Online, December 2005. http://martinfowler.com/eaaDev/EventSourcing.html Accessed February 7th, 2013.

[5] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. WebDSL: a domain-specific language for dynamic web applications. In *OOPSLA'08 Companion*, pages 779–780. ACM, 2008.

[6] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE'10 (Revised Selected Papers)*, volume 6563 of *LNCS*, pages 163–182. Springer, 2011.

[7] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI'90*, pages 234–245. ACM, 1990.

[8] Ralf Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[9] Ralf Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, 2004.

[10] Ralf Lämmel. Evolution of rule-based programs. *The Journal of Logic and Algebraic Programming*, 60–61(0):141–193, 2004.

[11] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *CHI'95*, pages 480–486. ACM/Addison Wesley, 1995.

[12] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP'08, pages 592–615, 2008.

[13] Ernst Lippe and Norbert van Oosterom. Operation-based merging. *SIGSOFT Softw. Eng. Notes*, 17(5):78–87, 1992.

[14] Alex Loh, Tijs van der Storm, and William R. Cook. Managed data: modular strategies for data abstraction. In *SPLASH Onward! '12*, pages 179–194. ACM, 2012. Research Track.

[15] Shahar Maoz, JanOliver Ringert, and Bernhard Rumpe. CDDiff: Semantic differencing for class diagrams. In *ECOOP'11*, volume 6813 of *LNCS*, pages 230–254. Springer, 2011.

[16] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, May 2002.

[17] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[18] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[19] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, October 2007.

[20] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *OOPSLA'06*, pages 451–464. ACM, 2006.

[21] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symb. Comput.*, 15(5-6):523–545, May 1993.

[22] Joost Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electron. Notes Theor. Comput. Sci.*, 200(3):3–23, 2008.

[23] Markus Völter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. 2013. http://dslbook.org/.

[24] Darcs Wiki. Everything you ever wanted to know about patch theory and conflictors. Online. http://repos.mornfall.net/public/darcs/DarcsWiki/PatchTheory.page.

[25] Wikipedia. Operational transformation — wikipedia, the free encyclopedia, 2013. [Online; accessed 7-February-2013].

[26] Klaus Wuestefeld. Prevayler. Online. http://prevayler.org/.

[27] Klaus Wuestefeld. Do you still use a database? In *Companion of OOPSLA'03*, pages 101–101. ACM, 2003.

[28] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE'05*, pages 54–65. ACM, 2005.

[29] Wuu Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.