Domain Modelling and Language Issues for Family History and Near-Tree Graph Data Applications

C.A. Maddra and K.A. Hawick
Computer Science, University of Hull, Cottingham Road, Hull HU6 7RX, UK.
Email: {c.maddra, k.a.hawick }@hull.ac.uk
Tel: +44 01482 465181 Fax: +44 01482 466666

April 2016

ABSTRACT

Domain-Specific Modelling is a powerful software engineering approach to building complex software systems. Domain-Specific Languages provide a powerful way of capturing and encapsulating the applications vocabulary and central ideas for whole families of software applications. We describe some domain-specific modelling approaches and techniques based around the application domain of family history or genealogy systems where the central data model is a near tree-like structure. We discuss how: querying; modification; and aggregation patterns of operation can be implemented in a number of ways for this domain application. We explore the scalability of the DSL approach and discuss wider issues in model and language development.

KEY WORDS
software design; tree data; genealogy applications; domain-specific languages.

1 Introduction

Domain specific modelling [23, 39] provides an important software engineering approach to formulating software designs based upon important features such as data models and key data structures for specific application areas. In this article we focus on the application area of family tree storage and analysis software.

Domain Specic Languages (DSLs) [13, 34] are little programming languages, that are often customizable [4] and are typically designed for use around exclusively around a chosen domain. Current popular general purpose languages such as C, C# and Java all focus on the implementation level [6, 18] of software at machine or virtual machine level. DSLs aim to provide what can be seen as a suitable compromise between hard to express concrete machine executions at implementation level and hard to execute vague human conversation at the solution level. The holy grail of a DSL project is to capture the essence of a family of applications within a DSL to allow separation of concerns from the implementation boilerplate and the solution creativity. This then allows multiple applications to be created without having to waste intellectual capital on the code intelligently automated by the DSLs back end. The separation can also allow for portability across different architectures through using different back ends.

Popular examples of DSLs include CSS, Regex, Flex/Bison, Blitz++, Make and JMock expectations. DSLs are not a mere set of libraries allowing you to plug other peoples algorithms into your solution at implementation level but a language which allows writing new algorithms and reusing others in the domains level of abstraction. This is code automation rather than just manual reuse. DSLs allow the user to work at the correct level of abstraction for writability, readability and maintainability. Writing at this level of abstraction captures the intention of the programmer rather than the byproduct of their intention in the form of an implementation [30].

The DSL approach has found uses in applications areas including: automatic structural optimisation for agent-based models [19]; operating systems development [7]; tile assembly in biological applications [8]; image processing [14]; wireless sensor nets [29]; and network systems configuration [36]. DSLs also find use more directly in development of software tools and software itself such as software version conversions [12]; code generation [24]; and program generation [27].

Generally DSLs are implemented as a user interface over a semantic model to maintain separation of the user interface and back end logic. A semantic model is an executable implementation of a domain generally in the form of one or more libraries implemented in a high level programming language. The DSL acts as the user interface to this semantic model, shielding the user from the technical implementation and allowing them to focus on the solution space. There may be many DSLs over a single semantic model allowing programmers to only concern themselves with the sub-domains they want to while retaining all code in a single base. Although there is no explicit extra functionality, a convenient way of populating the semantic model allows for more complicated systems to be created without concern of the underlying implementation. In other words, a DSL can allow the programmer to spend their intellectual capital on the problem domain rather than the solution space. The enhanced leverage on the underlying semantic model allows greater efficiency of thought and thus more functionality for the same work. This can be likened to the increase in productivity the manufacturing industry has seen from their progressive increase in automation and componentisation.

The definition of what makes a language a DSL is a blurry one, with languages being more or less of a DSL rather than a direct cut-off. This is especially true for internal DSLs which can be classed as a separate language to their host or merely a feature of it. The underlying point is not if a language is a DSL but if it can reduce the abstraction gap between the problem and solution domain compared to its general purpose counterparts. The successful use of DSLs relies on the mindset and practices of the programmers as much as the languages themselves. The way a language is used in a project can dene whether it is a DSL in that project or not as much as the features of the language.

There are a number of DSL tools and frameworks available [10, 11, 33]. Generally, a DSL generally conforms to this checklist of desirable features: 1) Clearly Identifiable Domain; 2) Limited Expressiveness; 3) Fluent Language Structure. It is helpful to look at DSLs though a concrete domain so patterns can be mapped directly to a set of related example implementations, which also allow us to show the compound nature of improvements in language. It is hard to display the advantages of domain specificity in a vacuum. Small changes in the level of abstraction can simply be mapped onto a library call or macro apparently removing the benefit of DSLs over GPLs, the advantage of these small changes in abstraction adds to a large change in the overall abstraction because it changes the level that the programmer needs to think. Working with macro's and library calls locks the programmer into the implementation level of a solution, and can quickly become clumsy in a comment littered solution. Domain specific solutions lock a user into thinking at the intention level rather than the implementation.

Our chosen domain of genealogy is suited to domain specific languages because it is well established with domain jargon which can be used in DSLs and is a firm domain which has guaranteed future work making a DSL worth the initial effort. Genealogical analysis is a new area which is becoming more important as we get more data and DSLs have a strong role to play in this as scientists realise it's not just about how much data you have but also the ease in creating novel ways to analyze this data. DSLs can be used as a method of input for genealogical analysis which can improve the quantity and quality of work possible given similar circumstances [26, 31]. Genealogy is a useful domain because it's a concrete application of directed graphs which link multiple genetic families together into a collection of blood lines forming tree-like graphs. Graphs are an exciting area for computer science because of their applicability as network oriented databases.

Genealogy file types can also be used to create single records of people which are not interlinked but stored merely for the recording of a persons information. For this report this is ignored as it is not important for genealogy research unless used to merge into trees and is a subset of the use of DSLs for full tree creation and modification.

We want to investigate the issues for using domain specific languages in graph like environments, with the concrete example of genealogy. Currently the genealogy area is filled with graphical applications to edit, view and share data because it is suitable for the layman market. We want to consider the useful techniques for and create a case study of the use of textual internal DSLs in this area to see what can be performed without the use of a graphical user interface.

Our article is structured as follows: In Section 2 we review background ideas on the genealogical application domain. We focus on technical aspects we have explored in our present project in Section 3 and in Section 4 we provide some case study details on our approach. In Section 5 we give a discussion of the implications of using Python and other tools for this sort of system and offer some conclusions and areas for further r work in Section 6.

2 Genealogy Domain Review

Genealogy is one of the oldest hobbies in the world. Whether people are drawn to it because of intrigue from a blank slate or tradition in the family technology is making finding, storing and sharing your ancestors far easier [3, 35]. Historically Genealogy has been done through paper based systems such as census data and birth records which are reasonably difficult to gain access to. With the advent of cloud oriented genealogy services such as ancestry.com and familysearch.org anybody can access these records to build their own family trees. These cloud oriented systems also remove the problem of scale as huge amounts of records can be searched through at a speed far faster than a human could search through even a few pages of paper records. Digital searching allows for complex searches including wild cards, conditionals and inferred missing data which are time consuming and difficult to perform by a human because of manual calculation and problems with remembering combinations of conditions.

Currently the genealogy market is dominated by on-line services because of the convenience of the cloud for storage and social networking. Big providers such as ancestry.org are active in improving the amount of data available to genealogists and the level of analytics which can be performed on this data [2].

One of the biggest offline suites is the gramps project, which is an open source python genealogy IDE which focuses not only on GEDCOM but also it's own Gramps XML format. The main concept behind Gramps' features is to allow the user to focus on their genealogical research in as much or as little detail as they want with peace of mind that it's all safely stored, searchable and sharable whenever they want. This is done using a GUI with limited options for extension, fitting its goal audience of hobbyist family tree researchers. Gramps is officially released for Linux with community supported windows and mac releases, this means different operating systems will give a different experience.

A long running genealogy suite is LifeLines which has been maintained as an open source project after the creator Tom Wetmore stopped working on it in 1994. Lifelines is based completely around GEDCOM but not any particular standard. It allows you to extend the default GEDCOM standard to suit your needs although this will cause comparability issues and data loss when sharing your genealogy data. Importantly for us lifelines pioneered the idea of using a report generation language to produce all the reports of the program rather than relying on preset report types. This DSL based approach allowed lifelines to perform any reasonable report generating task which could be imagined leaving it the choice of GenWeb and GenServ for their

backend This report generation language is covered further in similar genealogy DSLs.

GEDCOM (Genealogical Data Communication) is a specification designed to pass genealogical data between parties. It was developed by the Church of Jesus Christ of Latter-day Saints to aid genealogical research. GEDCOM is widely supported by genealogy software and has a simple reference based lineage-linked structure to link families together. [1].

For cross platform sharing to work the Genealogical Data Communication (GEDCOM) standard must be consistently followed. Sadly when big players such as Family Tree Maker do not follow the GEDCOM standard by adding in new tags it means software which uses GEDCOM has started to accept non-officially valid GEDCOM as good input to appease users. This makes it impossible to create a gedcom parser which will accept all 'GEDCOM' and clarifications on the subset accepted must be made. GRAMPS solves this problem with a third party add on called GEDCOM-extensions which supports pervasive changes to the gedcom standard. We focus on GEDCOM in this project because it is the current de-facto standard for sharing genealogical data.

The GRAMPS project setout with the goal of making a portable, machine and human readable format which can be read and written without data loss. The GRAMPS documentation alludes to XMLs compatibility with source control software when uncompressed and small file size when compressed. For performance reasons the XML representation is not used as the internal database for GRAMPS but as an export format. This format has not been used by this project because of it's limited compatibility with software suites aside from GRAMPS. Including GRAMPS XML as an extension to this project would be possible as the domain specific language layer would not need changing as the semantic models interface could remain the same.

The file format of Family tree maker, Ancestry.com's flagship software and claimed to be the number 1 selling genealogy software. The format is called FTW because Family Tree Maker was called Family Tree Maker for Windows in previous version. This format is proprietary and requires the Family Tree Maker software to convert to other formats. There have been complaints about the poor conversion to other formats as well [22]. This format is not suitable for this projects research because it's not open and it's associated tool is paid software aimed towards editing in a graphical setting.

Zandhuis presents the "Semantic web" as a way of storing genealogical data in an open and extensible way rather than the current file formats used. It presents a first attempt at a genealogical ontology to start the discussions for a standardized ontology with direct goals towards improving the current problems in exchanging genealogical data and automating it's processing. Integrity checks and intelligent processing can be performed on the genealogical data to check for constancy and potential errors. The ontology formalizes things such as events timing before, after or during a time period making automation of searching and analysis possible. This work could be complimented by a DSL which allows the input and manipulation of the semantic web in the same way this project deals with gedcom. Given that satisfactory APIs are provided by the ontology the DSL could give rich feedback based only on interfaces with the extensible ontology.

Displaying Genealogical data is an issue related to what the user needs to ascertain from the data. The traditional family tree is a generational graph starting from a single person as the root, but this method of notation does not show people in relation to time and poorly scales as descendants have families of their own causing excessive growth [25].

Mass scale genealogy is made possible by crowd sourcing many peoples research into combined files to aid everybody's research. The quick and successful merging of analogue genealogy data such as paper records relies on new copies being made and hand checking through existing records. This is tedious and error prone but has the advantage of interactions between researchers and knowledge known by the researcher but not stored in records may be combined with the existing information to aid merging or add new information. This process can be aided through digitized records because much of the tedious comparing work can be automated, leaving the researcher to merely confirm assumptions by the merging program. Unfortunately this can lead to errors in false-positive merges and false-false misses of merges which could be avoided by a human's intuition, errors due to incorrect records on either side are hard to avoid aside from common sense checks such as timeliness and location. The full automatic merging of digital files is also an issue, especially without strong standardization on what information is to be included in records (even if this requirement is just a method for saying if a field is not available) and how to format these records to make digital searching and sorting possible in all cases. Domain specific languages and enforced underlying models allows these to be enforced and inform the user where input does not meet the standard which is not possible in analogue or pure file-format input.

The semantic model idea allows the data to be stored in any format which matches the DSLs interfaces, this can be combined with other research areas such as work in aiding the automated process of using graph algorithms to merge family trees have been developed [38].

There are a number of established Genealogy DSLs available. We describe some of these and their features.

Life lines [37] has a reports language which you can specify any type of report you would like, with common reports programs given with the distribution as examples. This gives flexibility in comparison to the common solution of pre-made templates which you merely provide input to at the cost of the removal of simple GUI modifications of pre-made templates. The LifeLines language is implemented as many function calls in C. The decision to use C reduces the potential for internal DSL tricks as C has low language extension and tinkering support, this leaves the lifelines language close to the implementation level rather than bringing the user up to their solution level of abstraction.

This language has been used during undergraduate dissertation at the University of Hull but without success because of not being able to get to grips with the language referring to the simple examples and lack of an active online community. The documentation for the language is example based with the most common reports already having programs written up. Sadly the complexity of the reports language still makes it difficult for a novice to

understand how the reports are generated from the given code, this would be solved with an intention level approach. This example from the Lifelines language documentation [37] prints the ancestry of an individual. The individual is specified at runtime using the terminal.

```
proc main ()
  {
    getindi(indi)
    list(ilist)
    list(alist)
    enqueue(ilist , indi)
    enqueue(alist , 1)
    while (indi, dequeue(ilist)) {
      set(ahnen, dequeue(alist))
      d(ahnen) ". " name(indi) nl()
      if (e, birth(indi)) { " b. " long(e) nl() }
      if (e, death(indi)) { " d. " long(e) nl() }
      if (par, father(indi)) {
        enqueue(ilist , par)
        enqueue(alist , mul(2,ahnen))
      }
      if (par,mother(indi)) {
        enqueue(ilist , par)
        enqueue(alist , add(1,mul(2,ahnen)))
      }
    }
  }
```

Another DSL similar to our case study was written in 2013 by Paul Johnson [21]. This DSL is implemented in perl which has good support for extension by internal DSLs and Johnson has taken advantage of this to provide a good object oriented domain specific experience for the user. This DSL's distribution includes an unfinished 'lines2perl' program which converts lifelines programs into the DSL. This DSL fulfils the same role as the family tree manipulation DSL from this project and is extremely similar once the differences between a pearl internal DSL and python internal DSL are mitigated.

open a GEDCOM file and print out the names and birth dates of all individuals.

```
my $ged = Gedcom->new(
    grammar_version => "5.5",
    gedcom_file     => $gedcom_file,
    read_only       => 1);

for my $i ($ged->individuals)
{
  for my $bd ($i->get_value("birth date"))
  {
    print $i->name, " was born on $bd\n";
  }
}
```

3 Building Domain Specific Languages

The central idea of the DSL community is to create a language which works at the correct level of abstraction for a chosen finite domain, generally moving towards a more declarative environment to express solutions. Having a defined grammar for the domain means problems with overlap between domains, for example 'agents' having separate meanings for artificial intelligence and agent based modelling are explicitly dealt with as the domain of the language sets the context and semantics for the jargon within the language.

Writing languages in their interpreter allows us to perform interactive programming as is talked about in early 4th generational languages literature as monologue vs. dialogue [28]. With a concrete domain, 2-way conversations between the programming language and the programmer are useful because repercussions of a statement on an unknown dataset cannot be known before execution. An example of this is when asking for a persons mother, if they have more than one mother a simple question from the language run time environment will notify the user while allowing the syntax for regular events clean. In our case study this problem is solved by using mother() which selects the only available mother or if a conflict exists requests the user which they would like to use. If the user knows which mother they would like to use before performing the statement they can specify their chronological index or name.

How a DSL is related to it's host language denotes whether it's internal or external. Internal DSLs are hosted within an existing GPL and implemented using there language features. Internal DSLs are popular within the functional community because of the extensibility of languages like LISP and Ruby. Early languages such as C don't offer much support for internal DSLs because of their lack of extendable features. Recent imperative/-functional mixed languages such as SCALA and Python are a mix between these two extremes. The internal DSL code is generally mixed in with standard GPL code seamlessly and compiled or executed during the same process.

External DSLs are an explicitly different language to their host language which has it's own parser. As external DSLs have a separate parser to their host language they have complete control of their syntax and semantics rather than just what's afforded by the extensibility of the host. This extra control lends itself to domains where the model of execution or order of operations cannot be elegantly expressed using standard GPL syntax such as database querying. external DSLs can be intended for use as stand alone files e.g. configuration files or as explicit sections within GPL code eg regex.

An additional definition of "active libaries" [5] can be made for DSLs which although they are treated as internal DSLs they play a role in the compilation or execution of their code allowing for compile and/or run time domain specific optimization to be performed. Active libraries are popular with the extensible languages communities such as LUA and have been used in projects such as Husselmann's [20]. For this project we have created a set of internal python DSLs. Internal DSLs have been chosen because they allow seamless inter-operation between not only the DSLs themselves but also any other python code which is appropriate for the graph data created by the DSLs from the gedcom files. Generally speaking internal DSLs also have a lower implementation time to their external counterparts. Creating multiple languages each with a niche allows for a unique and appropriate view for each different sub-domain, layering these over the same semantic model helps code re-use.

In the traversal DSL, we use fluent interfaces for traversing the family tree. These allow a user to start at any person in the tree and move through their relatives in a single command. This is compatible with the use of run-time dialogue within the language as mentioned earlier so the fluent interface can be guided where they're are conflicts or multiple options.

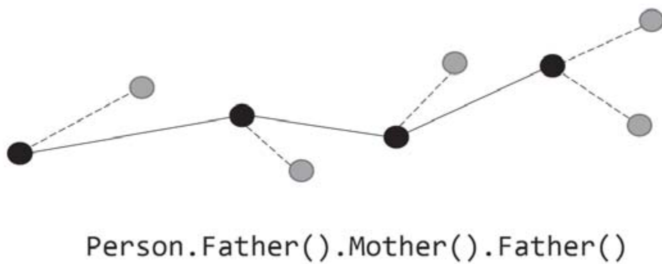

Figure 1: Fluent Interface to system.

It is considered good practice to separate the implementation of a DSL from it's front end [9]. Our DSLs are layered over a single semantic model because they are all based around the same software family and using the same boilerplate. Using the same backbone for each DSL allows code reuse and DSL inter operation without extra implementation time from re-writing code and creating interfaces. A semantic model is the semantics of a domain area in code, generally as a library or a set of libraries.

The semantic model is perfectly capable of being used by itself without the intervention of the DSLs, the DSLs are merely a level of abstraction above the semantic model allowing for better communication in that domain. We have chosen to have several DSLs over the same model so they can be different 'looking glasses' into the sub domains of our choice. Different users want to see different things from the same domain and this can be solved by using different languages for different purposes.

The downside of this is for people who wish to use multiple sub domains will have to learn multiple DSLs, although this is a direct trade with having to learn more language features from a single bloated language. There's an argument that by learning the principles large general purpose languages in general you learn the principles of how to program in any similar language.

Figure 2: System architectural model.

Figure 2 shows how Python can be integrated into the model architecture.As mentioned previously one of the benefits of using an internal DSL is access to the host language's existing ecosystem. Python is a particularly strong candidate for this because of the large amounts of scientific libraries and frameworks available such as Seaborn, Bokeh and Pygal for visualisation. Pre-existing tools to communicate with external languages and toolkits make this advantage even stronger because external interoperation with status-quo tools is dealt with out of the box. An example of this would be a user of our case study using Pymatlab to send the results of a query from our DSL into MATLAB for a pre-made script to produce graphs on the members connectivity.

4 Python Internal DSL

This project's back-end implementation expands on work by Nikola Skoric [32] (which itself was expanding on work by Daniel Zappala of Brigham Young University) who has created a GEDCOM parser which implements a subset of the GEDCOM 5.5 specification. Expanding on an existing GEDCOM parser saved time in picking the tags to implement and implementing them. This parser allows us to the the GEDCOM information and ingest it into an object oriented language representing it as a graph stored within a dictionary. This back end forms our semantic model of the family tree area, this semantic model can perform all our supported tasks in our domain even without the use of our front end DSLS.

The front-end DSLs have been created to allow sub-domains of genealogy to have a distinct DSL each which use the dominant data type of that sub domain as the basis for actions with subdomain specific jargon. Doing this allows us to work around the essence of each sub-area for example the traversal DSL is based around a tree and the family tree DSL is based around individuals and families. You can use the jargon of a family tree or data type tree in these DSLs interchangeably because the difference between in the DSLs is merely the users framing because the underlying representation is the same.

The DSLs are intended to be used within the python interpreter so the user can receive feedback as they program. The DSLs can also be used to create python scripts for re-use (for example doing common operations on multiple files at different times) or writing a new program or language as a level of abstraction above these DSLs. An example of a simple further level of abstraction over these DSLs such as a check box GUI are useful for people who do not wish to program at the expense of depth of expression. An interesting further level of abstraction would be a visual family tree manipulation language designed towards touch screens because it could provide intuitive manipulation of family trees which requires no computing knowledge at all.

As this project has been built on top of an existing GEDCOM parser project the handling of additional known and emerging data formats isn't ideal to implement. This implementation decision was an error at the start of the project which cannot be changed without re-writing the projects software. Using parsing technology such as Pyparse or funcparlib to filter GEDCOM data into a custom made, extensible model for the area of genealogy would have been a better solution because further work could be done with other back-end models and different input/output data types.

GEDCOM files store genealogy data using a lineage linked data format, this forms one or more graphs containing all the nodes. Order within the file itself is insignificant as the people and families are identified using reference tags.

The storage of the data in python is inherited from Nikola Sko-

ric's parser which this project builds upon. GEDCOM files are stored as individual lines which contain all the data and as containing objects for people and families which serve as encapsulating references.

5 Discussion

There are a number of advantages of using a DSL over that of using a model directly. Given that software reuse through a semantic model (in this case a framework) is beneficial because of the inherent benefits of software reuse we can ask: as a semantic model can be used directly, why is using a DSL beneficial. The theory behind using a DSL is to improve the programmers leverage on a certain domain, whereas the semantic model is merely designed to encapsulate business logic and facilitate software reuse. The separation of a back end model and a front end DSL aids the maintainability and scale-ability of the solution. This projects DSLs can provide a case study to this question.

Determining a qualitative measure such as leverage on a domain is a difficult task and has been tackled by many authors. A quantitative method of grading code is lines of code, they're issues with this method although for these purposes it is a viable indication [15–17]. The Lines of code metric can only be considered viable if both examples are written in good faith to be the most appropriate code for the task. For example not splitting statements into several lines to effect results or vice versa.

Traditionally genealogy data has been dealt with through word of mouth and paper records. The growing adoption of digital storage has opened up the question for how can we digitally store and manipulate genealogy data with at least the same amount of expression, compatibility and longevity as the previous paper records. To challenge this we must ask: where does paper excel and fail and how can a digital system improve upon the current situation.

Paper records are extremely flexible because the owner has complete control of what to write without needing any skills aside from reading and writing. This also leads to a problem because with flexibility comes increased chance for non-standardised, illegible or inconsistent records. Paper is also a long term storage solution depending on the type of paper used, environment it's stored in and the organisation of the storage system. (MORE)

The main candidates for digital user interfaces are:

- Direct file format editing e.g. GEDCOM
- Text based programmatic editing e.g. through a programming language
- Graphical editing e.g. manipulating a visualisation of a tree
- Form based applications e.g. GRAMPS
- Web based applications e.g. familysearch.org

These different strengths and weaknesses mean different audiences can benefit from different user interfaces. For example for quick viewing and editing of genealogy data applications are suitable for all audiences including those without programming knowledge, for more complex or repetitive tasks then a programming language can save time, effort and reduce mistakes. The benefit of the domain specific language over a semantic model approach we have taken in this project is different forms of user interface can all work on the same model. This means once you have spent the time developing the DSL developing graphical or form based interfaces above this DSL could be faster and cross-compatible with code written in the DSL.

6 Conclusion

In summary the intent-level communication lets the domain expert communicate in the language of the domain and their intention to be automatically moved to the implementation level by the DSL environment. We have found this reduces the time to implement manipulations, analysis and traversals of a GEDCOM file in relation to directly using the GEDCOM files or writing your own algorithms in python directly.

Using an internal DSL allows the project to nativity inter-operate with existing python libraries. This would be useful if for example the manipulation DSL was used with modifications performed by some existing graph analysis library.

The DSLs created for this project can be used as the backbone of future projects to reduce their workload. This goes with the extensible software philosophy of kernel and configuration being separate to allow malleable, reliable software. A future project in the genealogy DSL area could be a visual DSLs for modifying family trees. This project would be angled towards the applicability of visual DSLs where the domain especially lends itself to tactile manipulation and a case study for the use of a domain specific language to aid in the creation of another domain specific language in the same software family.

A possible future project could investigate the related area of graph-oriented DSLs, this could be considered a level of abstraction up from this project which focus a certain sub domain of graphs (family trees).The graph domain is exciting because of current developments in the graph database community bringing new, difficult problems to be solved into the space. Example projects could be graph manipulation languages, graph database languages and quantitative experiments based around these languages.

We believe the software engineering approach of developing DSLs that implement ideas embodied in models has considerable potential, and that this potential has not yet been widely exploited for many applications domains, particularly those with complex underpinning features.

References

[1] Allen, J.: Gedom(future direction) announced by family history (May 1998), https://listserv.nodak.edu/cgi-bin/wa.exe?A2=ind9805A&L=GEDCOM-L&P=R2&I=-3&T=0

[2] Baker, P.: How ancestry.com uses big data. Fierce Big Data (2014), http://www.fiercebigdata.com/story/how-ancestrycom-uses-big-data/2014-08-04

[3] Burton, J.: Genealogy issues paper. In: AITSIS Workshop on Genealogies (2002)

[4] Cong, J., Sarkar, V., Reinman, G., Bui, A.: Customizable domain-specific computing. IEEE Design & Test of Computers March/April, 6–14 (2011)

[5] Czarnecki, K., Eisenecker, U.W., Glück, R., Vandevoorde, D., Veldhuizen, T.L.: Generative programming and active libraries. In: Selected Papers from the International Seminar on Generic Programming. pp. 25–39. Springer-Verlag, London, UK, UK (2000), http://dl.acm.org/citation.cfm?id=647373.724187

[6] Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: Dsl implementation in metaocaml, template haskell, and c++. In: Domain-Specific Program Generation. pp. 51–72 (2003)

[7] Dagand, P.E., Baumann, A., Roscoe, T.: Filet-o-fish: practical and dependable domain-specific languages for os development. In: Proceedings of the Fifth Workshop on Programming Languages and Operating Systems. pp. 5:1–5:5. PLOS '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1745438.1745446

[8] Doty, D., Patitz, M.J.: A domain-specific language for programming in the tile assembly model. In: Proc. Fifteenth Int. Meeting on DNA Computing and Molecular Programming. pp. 8–11 (Mar 2009)

[9] Fowler, M.: Domain-Specific Languages. No. ISBN 0-321-71294-3, Addison Wesley (2011)

[10] de Geest, G.: Building a framework to support Domain-Specific Language Evolution using Microsoft DSL Tools. Master's thesis, Software Engineering Research Group, Delft University of Technology (2008)

[11] de Geest, G., Savelkoul, A., Alikoski, A.: Building a framework to support domain-specific language evolution using microsoft dsl tools. In: Proc. 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07) (2007)

[12] de Geest, G., Vermolen, S., van Deursen, A., Visser, E.: Generating version convertors for domain-specific languages. In: Proc. 15th Working Conf. on Reverse Engineering (2008)

[13] Ghosh, D.: Dsl for the uninitiated - domain-specific languages bridge the semantic gap in programming. Communications of the ACM 54(7), 44–50 (2011)

[14] Guenter, B., Nehab, D.: Neon: A domain-specific programming language for image processing. Microsoft Tech Report MSR-TR-2010-175, Microsoft Research (2010)

[15] Hawick, K.A.: Engineering domain-specific languages for computational simulations of complex systems. In: Proc. Int. Conf. on Software Engineering and Applications (SEA2011). pp. 222–229. No. CSTN-123, IASTED, Dallas, USA (14-16 December 2011)

[16] Hawick, K.A.: Engineering internal domain-specific language software for lattice-based simulations. In: Proc. Int. Conf. on Software Engineering and Applications. pp. 314–321. IASTED, Las Vegas, USA (12-14 November 2012)

[17] Hawick, K.A.: Fluent interfaces and domain-specific languages for graph generation and network analysis calculations. In: Proc. Int. Conf. on Software Engineering (SE'13). pp. 752–759. IASTED, Innsbruck, Austria (11-13 February 2013)

[18] Hemel, Z.: Methods and Techniques for the Design and Implementation of Domain-Specific Languages. Ph.D. thesis, Delft University of Technology (2012), iSBN 978-90-8570-794-3

[19] Husselmann, A.V., Hawick, K.A.: Automatic high performance structural optimisation for agent-based models. In: Proc. 14th Int. Conf. on Software Engineering Research and Practice (SERP'14). pp. 1–7. WorldComp, Las Vegas, USA (21-24 July 2014), http://www.hull.ac.uk/php/466990/csi/reports/0010/csi-0010.html

[20] Husselmann, A.: Data-Parallel Structural Optimisation in Agent-Based Modelling. Ph.D. thesis, Computer Science, Massey University, Albany, North Shore, New Zealand (May 2014)

[21] Johnson, P.: Gedcom 1.19 (August 2013), https://metacpan.org/pod/Gedcom

[22] Jones, T.: Ftw gedcom (March 2009), http://www.tamurajones.net/FTWGEDCOM.xhtml

[23] Karlsch, M.: model-driven framework for domain specific languages demonstrated on a test automation language. Master's thesis, Hasso-Platner-Institute of Software Systems Engineering, Potsdam, Germany (2007)

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

[25] Kim, N.W., Card, S.K., Heer, J.: Tracing genealogical data with timenets. In: Proceedings of the International Conference on Advanced Visual Interfaces. pp. 241–248. ACM (2010)

[26] Ledford, H.: Genome hacker uncovers largest-ever family tree. Nature (October 2013), http://www.nature.com/news/genome-hacker-uncovers-largest-ever-family-\\tree-1.14037

[27] Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.): Domain-Specific Program Generation. No. 3016 in LNCS, Springer (2003), ISBN 3-540-22119-0

[28] Martin, J.: Fourth Generation Languages: Principles. Pretice Hall (1985)

[29] Sadilek, D.A.: Prototyping and simulating domain-specific languages for wireless sensor networks. Tech. rep., Humboldt-Universitat zu Berlin, Institute for Computer Science (2007)

[30] Simonyi, C.: The death of computer languages, the birth of intentional programming. Tech. rep., Microsoft Research (1995)

[31] Singer-Villalobos, F.: Computer scientists at ut austin crack code for redrawing bird family tree. Texas Advanced Computing Center (2014), https://www.tacc.utexas.edu/-/computer-scientists-at-ut-austin-crack-code\\-for-redrawing-bird-family-tree

[32] Skoric, N.: simplepyged (February 2014), https://github.com/dijxtra/simplepyged

[33] Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages and Computing 15, 291–307 (2004)

[34] Taha, W.: Domain-specific languages. In: Pro. Int. Conf. Computer Engineering and Systems (ICCES). pp. xxiii – xxviii (25-27 November 2008)

[35] Veale, K.J.: A doctoral study of the use of internet for genealogy. Historia Actual Online 7 pp. 7–14 (2009)

[36] Voellmy, A., Agarwal, A., Hudak, P., an Sam Burnett, N.F., Launchbury, J.: Don?t configure the network, program it! domain-specific programming languages for network systems. Tech. Rep. YALEU/DCS/RR-1432, Yale University, USA (July 2010)

[37] Wetmore, T.: The lifelines programming subsystem and report generator (2005), http://lifelines.sourceforge.net/manual.3.0.39/ll-reportmanual.html

[38] Wilson, R.: Graph-based remerging of genealogical databases. In: Workshop on Technology for Family History and Genealogical Research. vol. 1 (2001), http://dagwood.cs.byu.edu/fht/workshop01/fht2001prog.php http://dagwood.cs.byu.edu/fht/workshop01/final/Wilson.pdf

[39] Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: Domain-specific metamodelling languages for software language engineering. In: Proc. Software Language Engineering (SLE 2009). LNCS, vol. 5969, pp. 334–353 (2009)