Supporting Domain-Specific Programming in Web 2.0: a Case Study of Smart Devices

Navid Ahmadi
Faculty of Informatics
University of Lugano
Lugano, Switzerland
navid.ahmadi@usi.ch

Francesco Lelli
Faculty of Informatics
University of Lugano
Lugano, Switzerland
francesco.lelli@usi.ch

Mehdi Jazayeri
Faculty of Informatics
University of Lugano
Lugano, Switzerland
mehdi.jazayeri@usi.ch

***Abstract*—Web 2.0 communities emerge regularly with the growing need for domain-specific programming over Web APIs. Even though Web mashups provide access to Web APIs, they ignore domain-specific programming needs. On the other hand, developing domain-specific languages (DSLs) is costly and not feasible for such ad hoc communities. We propose *User Language Domain* (ULD): an intermediate Web-based architecture using a domain-specific embedded languages approach that reduces the cost of DSL development to plugging the Web APIs into a host end user programming language. We have implemented the proposed architecture in the context of smart devices, where we plug the functionality of different Lego Mindstorms devices into a Web-based visual programming language. We expect that several domains, such as smart homes or wearable computers can use the ULD architecture to reduce development effort.**

***Keywords*-domain specific languages; Web 2.0 communities; end user programming; plugin architecture; smart devices; ubiquitous computing;**

I. Introduction

The Web has become the main platform for the information society. Lots of information and functionality have been exposed on the Web as Web Services or Rest APIs, encouraging domain-specific application libraries on top of which domain-specific applications have been built. The growth of this trend has led to envisioning the Web of applications[1]. Relying on the Web-based socio-technical environment, Web 2.0 communities have emerged[2]. These communities consist of mostly end users sharing interest that falls in one or many problem domains. The Web-based socio-technical environments provide hypertext-based collaboration. But to solve the computational problems, they need access to domain-specific programming tools. Web Mashup tools[3] provide access to these services mostly for aggregation and visualization of information, independently of the domain knowledge. Yet end users as the experts in their own domain need support for domain-specific application development[4]. Currently, Web 2.0 communities are limited to the tools and applications provided by professionals to exploit provided services.

Web 2.0 communities address different problem domains and are varied in their programming needs. Providing a one-off programming tool and language (e.g., mashup tools) does not fulfill the needs of all the communities. Each community needs its own domain specific language to capture the recurring problems of its domain. Yet, the domain engineering process to develop a particular DSL is costly and implementation of the underlying compiler or interpreter needs professional programming expertise. These barriers are in contrast with the casual and voluntary spirit of Web 2.0 communities. Moreover, Web 2.0 communities usually emerge as a situational interest rather than in formal and organizational settings. Design and implementation of a DSL from scratch is not feasible in such dynamic contexts.

Domain-specific embedded languages (DSEL) have been exploited to lower the barrier of developing a DSL from scratch by reusing a general purpose language (GPL) to access a domain-specific application library[5][6][7]. The same approach can address the computational needs of emerging communities, i.e., an emerging community of end users may easily choose the required services that capture the domain functionality and embed them into an existing end user programming language that meets best the problem domain. As a result, the host end user programming language provides access to the main functionality of a domain; hence the cost of compiler development reduces to the cost of embedding the language for accessing the provided application library. The services available on the Web form a pool from which primitive domain operations may be selected.

In this paper we suggest an intermediate (architectural) solution between one-off domain-independent tools such as mashup tools and costly domain-specific languages. Accordingly, we devise *User Language Domain* (ULD): a Web-based architecture to support domain-specific programming based on DSELs in Web 2.0 communities. ULD requires as a host language a general-purpose end user programming language with a plugin architecture. Plugins are used for embedding new functionality that captures the programming model of the domain. We have applied the architecture in

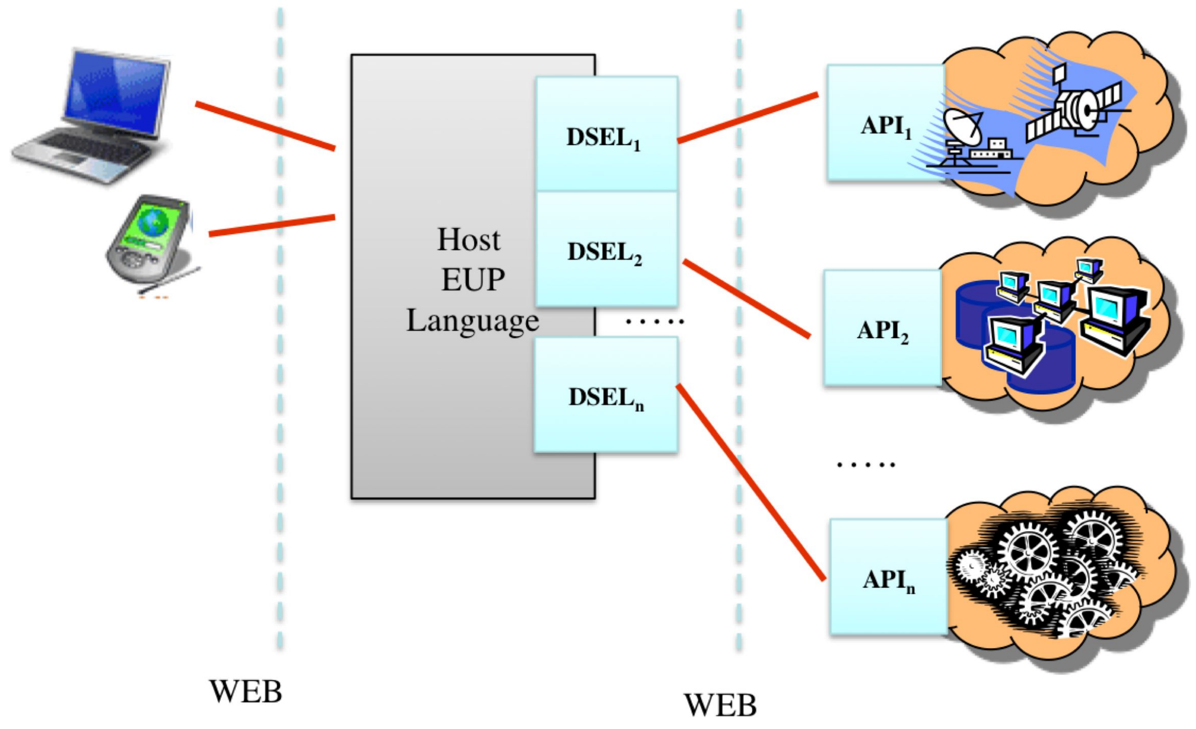

Figure 1. Domain-specific embedded language approach to enable domain-specific programming on the Web

the context of smart devices [8][1]. Each device represents a separate domain with its own functionality and exposes its instructions on the Web as Web services. A Web-based visual programming language with a plugin architecture enables the exposed instructions to be plugged into the programming environment so that users are enabled to write interactive programs for their devices. Our approach clearly separates device specific, domain specific, and end-user programming activities.

The rest of the paper is organized as follows: in Section II we sketch the ULD architecture to support situational domain appropriation; In Section III we report on an implementation case study in the context of smart devices; In Section IV we describe the related work while Section V discusses our experience and presents the design tradeoff. Finally in section VI we draw some conclusion.

II. ARCHITECTURE

In this section we present a Web-based architecture for enabling domain-specific programming in the dynamic ecosystem of Web 2.0 communities. Our idea is based on using DSELs to provide programming tools using a host end user programming language. We assume that the application domain libraries are accessible on the Web, as in practice hundreds of APIs are already shared and accessible on the Web[9]. Our approach has the following advantages:

- The DSL development cost is reduced to embedding the library into a host end user programming language, addressing the dynamic ecosystem of emerging Web 2.0 communities.
- The DSEL approach provides domain experts with domain-specific programming structures that capture recurring problems of the domain, rather than providing them with one-off domain-free tools such as mashups.
- The end user programming language facilitates the programming for domain experts by relieving them of low-level programming tasks[10]

Figure 1 shows a general overview of the ULD architecture. Different domains expose different functionality as API on the web. Examples of such APIs include access to social networking (such as the facebook API), weather forecasting, and APIs to access devices. For each API we create a DSEL using the host EUP language which allows domain experts to write their domain-specific programs using the provided API.

This approach is formalized in Figure 2, where we propose a Web-based architecture for separating the functionality from the programming language. According to this architecture, the programming language communicates with multiple application domain libraries. Each application domain consists of multiple instructions that will be

[1]in this context ULD can also be considered as *User Language Device* since we are applying the architecture to the domain of smart devices

embedded into the programming language to enable end users to compose these instructions. One or more domain adapters are used for communicating with the remote services. The programming language, instruction plugins, and service adapters are located at the client side, while the domain functionality is located on one or more servers that are accessible through their URLs.

We assume that users know the location of the services that they want to use. Once the URL is selected, the Domain Adapter module connects to the service and acquires the list of operations of the service. The Application Domain library can automatically generate the instruction plugins from the signature of the operations and plug this new functionality directly into the programming language.

When an instruction is called from inside the programming language, the service adapter serializes the method call into the format specified by the server and sends the instruction to the service. The service processes the instructions and returns back the result to the adapter. Finally, the adapter passes the results to the programming language environment.

From Figure 2 we can also note that the End User Programming Language module can use multiple Application Domain Libraries (ADLs) for building its instructions. Each ADL can use multiple *Instruction Plugins* that are built using multiple *Domain Adapters* that wrap a particular *Domain Functionality*.

Examples of domain adapters include: (i) access to databases, (ii) Access to software as a service[11] and (iii) Access to devices that expose their functionality to the Web using, for example, Web or Rest services. In addition, in case of data hardcoded inside one or more HTML pages without any defined format, Web scraping methods can be exploited to extract the information.

The programming language has to provide a programming paradigm which is suitable for programming the domain functionality, while offering high-level programming for end users. We adopt an end user programming language in the class of empowering systems[12], which provide Turing-complete programming power and enhanced visual techniques to assist the end user in programming.

III. CASE STUDY

We have applied the ULD architecture in a case study with smart devices[8]. In this scenario, users interact with devices over a computer network. The interaction between users and devices usually takes place through a single-purpose user interface that enables a particular functionality. Most devices such as home appliances and sensors are independent of the programming paradigm and as long as their functionality is accessible by the programming language, they can be programmed in any programming paradigm.

Using the described architecture we separate the functionality of devices from the programming language. In this way the functionalities of each device can be embedded in any

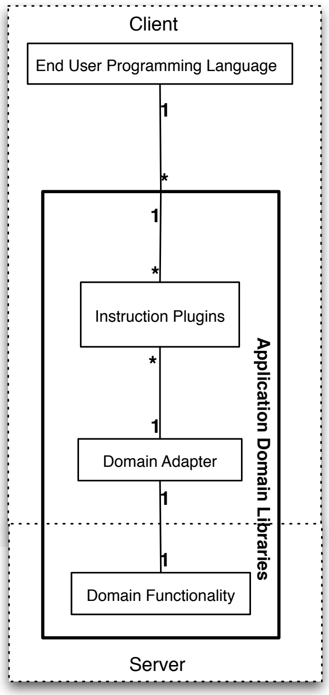

Figure 2. User Language Domain (ULD): A Web-based Architecture for embedding the application domain libraries into the host programming language

programming language. In addition, each device can provide some introspection for automating the access to its methods. The rest of this section is divided in 2 parts: first, we describe how we implement the domain functionality, then we describe the proposed programming language showing how we embed the instructions based on the functionality provided by our devices.

A. Application domain

We evaluate the ULD architecture by implementing two different configurations of Lego Mindstorms[2]. These devices have multiple sensors and actuators. By building different configurations of the robot we have emulated different specific domains while reusing some part of the developed code across different configurations. Moreover, multiple sensors and step motors make the Lego Mindstorms a complex device that offers more sophisticated functionality than other smart devices such as home appliances like a TV, microwave, or referigerator. Figure 3 shows the implemented architecture for each device.

We use the lejos library[3] as the operating system of Lego Mindstorms. Lejos also provides an API for remote accessing of different bricks such as sensors and motors of the Lego Mindstorms. A bluetooth connection is used for

[2]http://mindstorms.lego.com
[3]http://lejos.sourceforge.net/

accessing the device from a nearby server. Each device is paired with a server that exposes a Web based API offering device functionalities. We use the Instrument Element[4] [13], [14] for providing the same service while implementing the different functionalities of the device.

We used two different Lego Mindstorms configured and assembled in different configurations. Figure 3 shows the implemented architecture for each device. Inside the Mindstorms three different modules run in parallel.

- A feedback loop performs local actions such as stopping the step motors if one or more touch sensors are triggered or if the ultrasonic sensor detects objects that are too close.
- The communication program module initiates and maintains communication with the server.
- The control logic receives commands from the communication program and triggers the required action in the hardware. It also periodically senses the values of the sensors and sends asynchronous update messages to the server.

An Instrument Element runs on the server and offers the Web APIs by implementing the following modules:

- The NXT APIs is a module that wraps the low level communication library of Lejos offering a higher level interface.
- The Feedback loop receives commands implementing recovery action depending on asynchronous messages coming from the device. Examples include maintaining a paired connection with the device and warning the user in case of low battery.
- The Cloud Access module fetches the sensor data coming from the device and stores them in a remote storage.

Finally the high-level functionalities that are offered as Web API for the first device are:

- move-backward *speed*
- move-forward *speed*
- stop
- turn *X* degree
- sense *sensor name*

While for the second device:

- move-forward *speed*
- stop
- turn *X* degree with radius *Y*
- sense object at your *direction*

We note that some instructions are common for these devices while others are different. Together with these commands, we also provide introspection functionality for the automatic discovery of the available instructions.

[4]available at http://instrumentelem.sourceforge.net

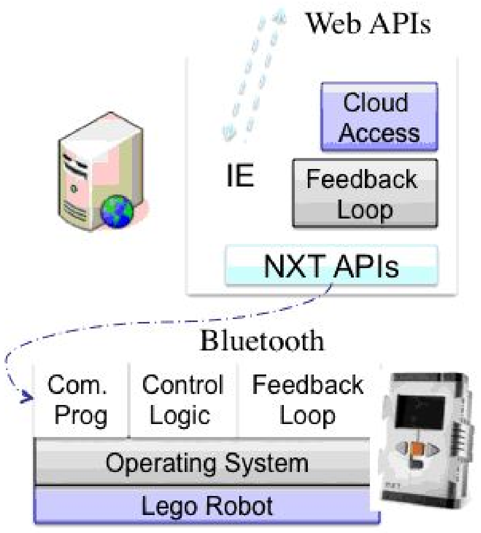

Figure 3. ULD Architecture for Programming the Lego Mindstorms

B. Programming Language

We have selected a visual rule-based programming language to program Lego Mindstorms. This language, together with its underlying compiler, support an event-based programming paradigm, conforming to JavaScript's programming paradigm. Each event is a method that can be programmed by the user and consists of multiple rules. Each rule is composed of conditions and actions that are instances of instructions. Each instruction can be either a generic instruction that comes with the programming language, such as setting the value of a variable, or a domain specific instruction plugin that is embedded into the programming language. At programming time, the composition of conditions and actions is supported by dragging a condition/action from the conditions/actions list and dropping it into the rules. The program written by the end user is serialized to XML format. Then the underlying compiler compiles the program to native JavaScript application. Figure 4 shows a snapshot of the programming environment.[5]

The figure shows a program that consists of two rules. The first rule measures the battery level and in case of low battery it turns the device off. Otherwise, the second rule will be tested according to which if the ultrasonic sensor of the device senses an obstacle in front of it, the device will turn 90 degrees and move forward with the specified speed for a while and then it will stop.

In the rest of this section we describe how the instruction plugins are implemented and how we automate the plugin generation using the introspection methods provided by the Web API of the devices.

[5]The programming environment has been under development as an open-source project available at http://weup.sourceforge.net

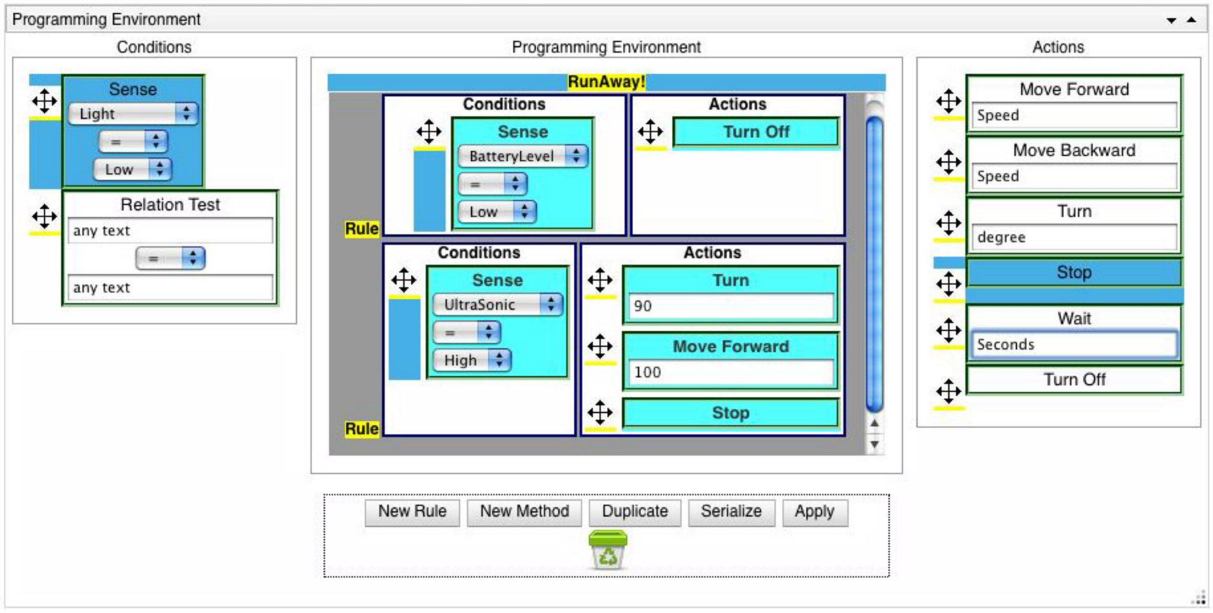

Figure 4. Visual Programming Environment

1) instruction plugins: The programming language has a plugin architecture. All the instructions are defined in the XML format. The definition of an instruction contains the list of instruction arguments, visual representation of the instruction, and the execution code. Figure 5 demonstrates the definition of a general instruction of the programming language, called Relation Test, that evaluates two expressions and applies a relational operator on them. This instruction is defined as a condition to be listed in the list of condition instructions. The definition consists of three main parts.

- *InputList* element contains a list of instruction input arguments. Each input has an id and a type. In this example, three inputs are defined: two text inputs for entering the expression and one relational operator.
- *VisualRepresentation* element contains the HTML-based user interface definition of the element. The programming language extracts this HTML definition, creates a DOM tree out of it and attaches it to the DOM tree of the programming environment whenever required. In this example, a table element has been used to show the instruction name and three input arguments in a column. Each input argument is also defined as a XML-based plugin and contains its own UI to be instantiated and attached to the programming environment (described later).
- *ExecutionCode* element contains an execute method that has to be implemented for each instruction. The execute method will be executed at run time. In this example, the execute method gets the value of all three input arguments, applies the chosen operation on two expressions using JavaScript eval() function and returns the value.

```
<?xml version="1.0" encoding="utf-8"?>
<!--Visual Instruction Definition-->
 <VID name="Relation Test" type="condition">
     <InputList>
          <input id="input1" type="Text"></input>
          <input id="input2" type="RelationalOperator"></input>
          <input id="input3" type="Text"></input>
     </inputList>
     <VisualRepresentation>
          <table id="instruction" draggable="true" type="condition">
               <tr><td id="instructionName">Relation Test</td></tr>
               <tr><td><input1 id="instructionInput"/></td></tr>
               <tr><td><input2 id="instructionInput"/></td></tr>
               <tr><td><input3 id="instructionInput"/></td></tr>
          </table>
     </VisualRepresentation>
     <ExecutionCode>
       this.execute=function(){
          //test two expressions against the operator
          //and return the result
       }
     </ExecutionCode>
</VID>
```

Figure 5. Visual Instruction Definition for a Relation Test Instruction.

Each instruction is composed of zero or more input arguments, and each input argument has a data type. To reuse the data types, we have also implemented them as plugins, i.e., each data type is defined in the XML format. Figure 6 shows the XML content of a defined datatype. Each visual datatype defines a HTML-based visual representation that is instantiated and attached to the DOM tree of the programming environment whenever an instruction containing an input argument of such datatype is instantiated. All visual datatypes have to implement two methods called getValue

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Visual Datatype  Definition -->
<VDT name="RelationalOperator">
  <span id="VisualRepresentation">
    <select name="RelOpList">
    <option value="=="> = </option>
    <option value="!=">!= </option>
    <option value="&gt;">&gt;</option>
    <option value="&gt;=">&gt;=</option>
    <option value="&lt;">&lt;</option>
    <option value="&lt;=">&lt;=</option>
    </select>
  </span>
  <setValue>
    this.setValue=function(value){
      //choose an operation in UI
      //based on the input argument
    }
  </setValue>
  <getValue>
    this.getValue=function(){
      //return the selected element
      //as a text value
    }
  </getValue>
</VDT>
```

Figure 6. Visual Datatype Definition for a Relational Operator.

and setValue that are used both for serialization of the program written by end user to an XML-based document and by the execute method of the instruction at run time.

2) Automatic plugin generation: If the application domain libraries can provide an introspection mechanism a plugin-based programming language can generate automatically the instruction for a new library when the new functionality is being added to the programming environment using its URL. In the context of this case study, each device exposes a different set of commands to the Web so that each Mindstorms has to be treated as a different application domain library. To access the device commands from inside the programming language, we have to create an instruction plugin for each command of the device. In our implementation, the service adapter component mentioned in the architecture is responsible for both communication and generation of instruction plugins. The instruction plugins are being generated automatically when a new Lego Mindstorms is connected to the programming language by providing its URL to the programming environment. The end user only provides the Web service URL to the programming environment. Then the programming environment acquires a list of device commands from the device, generates instruction plugins for each single command, and adds the instructions to the programming language.

To generate the instruction plugins on-the-fly, we created a template for the definition of instructions of the programming language, as shown in Figure 7. The template contains

```
<?xml version="1.0" encoding="utf-8"?>
<VID name="__commandName" type="__commandType">
  <!--Visual Instruction Definition Template-->
  <inputList>__inputList</inputList>
  <VisualRepresentation>
    <table id="instruction" draggable="true"
     type="__commandType">
     <tr>
       <td id="instructionName"
       align="center">__commandName</td>
     </tr>
     __inputUI
    </table>
  </VisualRepresentation>
  <InstructionCode>
    this.execute=function()
    {
      var params= new Array();
     __params
     ExecuteCommand("__commandName",params);
    }
  </InstructionCode>
</VID>
```

Figure 7. Plugin Template.

the generic definition of an instruction similar to the one shown in Figure 5. When a Lego Mindstorms is connected to the programming environment, the service adapter (described in the architecture) communicates with the Web service exposed by Instrument Element. The Instrument Element provides a getCommands operation that returns a list of commands of the device, input list of each command, and the type of each input. The service adapter retrieves the list of commands and for each command instantiates a plugin template and fills it with the command's parameters such as the command name and its input list with proper data types of the programming language for each input. It also modifies the execute method to call the ExecuteCommand method with the input values coming from the programming language. The ExecuteCommand method which is implemented in the service adapter invokes the device command with its parameters and returns the result during the execution.

IV. Related Work

Domain specific languages for domain experts Van Deursen and Visser[6] define a domain-specific language (DSL) as ”a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.” Fischer et al. [15] discuss the evolutionary nature of domain modeling and construction that is done by a community of practice over time, and suggest supporting collaboration in domain construction. Domain-Oriented Design Environments have been suggested by Fischer et al[16] to support human problem-domain interaction rather than human computer interaction. End user programming environments such as

AgentSheets have been developed to enable domain experts to build their own DSLs[17].

With the emergence of Web 2.0, the Web has become a social platform on which new communities emerge constantly. Web 2.0 communities need to develop applications in their own domain of interest, i.e., shifting the Web towards the Web of applications[1]. Yet the ecosystem of Web 2.0 communities is highly dynamic and developing DSLs for these communities is not feasible. Fischer et al. provide guidelines on supporting domain experts in developing software artifacts, particularly with respect to the existing growth of the end user developers on the Web [4]. Mernik et al. observe that the development of a DSL rarely goes further than developing an application library, since DSL development requires both domain knowledge and language development expertise[5]. Application libraries used together with a general-purpose programming language are called domain-specific embedded languages[7]. Developing a DSEL reduces the the programming language development process of DSL to developing the programming structures of the domain in a host GPL. The UDL architecture reflects the same approach of plugging a domain application library to a host end user programming language.The domain adapter component in the UDL architecture is responsible for capturing the domain-specific constructs as high-level instructions over the provided domain functionality and plugging them as instructions of an end user programming environment.

Programming ubiquitous devices In recent years the idea of programming ubiquitous devices has gained increasing attention. In [18], Truong et al. propose a programming language based on the The Magnetic Poetry Metaphor. Users arrange poetry in a virtual blackboard that results in a sequence of actions of a set of devices. In [19] the authors propose different approaches for programming devices including textual and visual languages for advanced users. Blackwell and Hauge provide an architecture for programmable home appliances[20]. For the sole domain of Lego Mindstorms that we have used in our case study, Microsoft suggests the *Microsoft Robotics Developer Studio*[6] which consists of a visual programming language to program the device and access to the device functionality from other Microsoft programming tools. Microsoft Robotics Developer Studio is an example of domain-specific language developed for the single domain of robotics, while our approach addresses different domains and devices by embedding them into a suitable programming language.

Web Mashups The demand to combine data and services on the Web according to the user's need has led to the development of Web mashups[3]. Each mashup is a combination of data and services from several sources on the Web. The data is usually provided as a Web feed or the output of Web or REST services. Mashups are usually native Web applications written in JavaScript and using AJAX, JSON and XML-based technologies. ProgrammableWeb.com has become the main stream for sharing mashups and Web APIs.

To lower the barrier of writing a Web mashup, mashup tools have been created mostly by commercial companies yet freely accessible on the Web. These atrifacts usually provide a higher level of abstraction for the programming of mashups. For example Yahoo! Pipes [7] provides a GUI to pipe data to operators such as aggregators and services. Google Mashup Editor (GME) [8] provides a high level mashup development API to be used in JavaScript. Similar mashup tools with slightly different approaches such as Microsoft Popfly and Intel Mashmaker have been offered to mashup developers[21]. Domain-specific languages for building mashups have been developed [22][23]. End user programming approaches such as data-flow programming, scripting, and spreadsheets have been applied in mashup development [24], [25], [26], [27].

The fast growth of Web mashups shows the acceptance of building applications on top of existing data and APIs. Mashup building tools present the power of existing technology to build Web-based applications using data feeds, AJAX, JavaScript, and REST APIs. However, existing mashup development tools provide a one-off programming language for all the existing APIs on the Web. These tools ignore the need for domain-specific languages and appropriate programming structures specific to different application domains and their corresponding APIs. In contrast, we believe that although the functionality is suggested as APIs on the Web, providing tools to just communicate with these APIs and aggregate the results is not enough for domain experts of a domain to build their own applications over the provided functionality. Our architecture wraps the domain-specific structures of each application domain in the domain adapter and embeds it into an end-user programming language to create a domain-specific programming language on the Web.

V. Discussion

Our case study of smart devices helps to show the promise of separating functionality from the end user programming language in end user programming. Currently, web based artifacts are released together with a set of open APIs that can be used by their customers for building extensions. Our solution allows an efficient way of embedding these APIs into a customizable wed based end user programming environment.

Based on this case study we note that:

- The idea of adopting a plugin architecture introduces a tradeoff between (i) having a standardized and uniform

[6]http://msdn.microsoft.com/en-us/robotics/default.aspx

[7]http://pipes.yahoo.com

[8]http://editor.googlemashups.com

access to different APIs and (ii) having a completely free set of methods. In the first case the realization of the Application Domains Library is simpler and portable. However, the introduction of domain dedicated features is more complex.
- In the realized prototype we have considered the possibility of separating the functionality from the visualization. However, we could also embed in the APIs a set of metadata for the visualization of the instructions and this will result in a more uniform adoption across possible domain languages.
- The end user programming language developed in this case study runs completely on the client side of the Web browser. By using JavaScript as the assembly language of the Web, the underlying compiler of the programming language compiles the visual language into the complete JavaScript application. Since JavaScript is an event-based programming language, the compiler is forced to map the visual programming language to an event-based language.
- In our case study, the provided visual language consists of methods that can be treated as events and be triggered upon calling them; hence it is straightforward to map this language to an event-based language. However, in case of need for other programming paradigms, such as functional programming or data-flow programming, the mapping to an event-based language is more complex.
- Server-side execution of end user developed applications creates a trade-off between flexibility and interactivity. Although server-side compilation and execution increases the flexibility on possible technologies and methods to be used for compilation and execution, it hampers interactivity during application execution, which is important in domains such as scientific simulations and games. A hybrid solution between client- and server-side execution has to be chosen according to the domain.

VI. CONCLUSION

In this paper we propose ULD, a Web-based architecture to separate the domain functionality from the programming language to support programming in the highly dynamic ecosystem of Web 2.0 communities. We devise a plugin architecture for an end user programming language to be used as a host language into which the functionality of multiple domains can be plugged. We apply our architecture in the context of smart devices. As a result, end users are provided with a visual programming environment to program different Lego Mindstorms devices. The visual programming interface is automatically generated from the device functionality and embedded into the programming environment.

The adoption of the Web for end user programming opens new frontiers for collaborative developments. The developed case study will serve as a testbed where we will evaluate different collaborative end user programming tools and methods in order to assess the potential of end user programming in online social environments and understanding their challenges.

ACKNOWLEDGMENT

The authors would like to thank Alimur Rashid, Sohrab Ameli, and Krzysztof Zawadzki for their contribution in the development of the case study.

REFERENCES

[1] T. Raman, "Toward 2^w, beyond web 2.0," *Communications of the ACM*, vol. 52, no. 2, pp. 52–59, 2009.

[2] T. O'Reilly, "What is web 2.0: Design patterns and business models for the next generation of software," 2005. http://oreilly.com/web2/archive/what-is-web-20.html.

[3] R. Yee, "Pro Web 2.0 mashups: Remixing data and Web services," *Apress*, 2008.

[4] G. Fischer, K. Nakakoji, and Y. Ye, "Metadesign: Guidelines for supporting domain experts in software development," *Software, IEEE*, vol. 26, no. 5, pp. 37 – 44, Sep 2009.

[5] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[6] A. V. Deursen and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices*, Jan 2000.

[7] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, no. 4es, Dec 1996.

[8] M. Weiser, "The computer for the 21st century," *Scientific American*, pp. 66–75, 1991.

[9] http://www.programmableweb.com

[10] B. Nardi, "A small matter of programming: Perspectives on end user computing," *MIT Press*, Jan 1993.

[11] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro, "Service-based software: the future for flexible software," *Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, pp. 214–221, 2000.

[12] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2, pp. 83–137, Jun 2005.

[13] F. Lelli and C. Pautasso, "The tiny instrument element," *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, pp. 293–304, 2009.

[14] F. Lelli, E. Frizziero, M. Gulmini, and G. Maron, "The many faces of the integration of instruments and the grid," *International Journal of Web and Grid Services*, vol. 3, no. 3, pp. 239–266, Jan 2007.

[15] G. Fischer, S. Lindstaedt, J. Ostwald, M. Stolze, T. Sumner, and B. Zimmermann, “From domain modeling to collaborative domain construction,” *DIS ’95: Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*, Aug 1995.

[16] G. Fischer, “Domain-oriented design environments: supporting individual and social creativity,” *Computational Models of Creative Design IV*, pp. 83–111, 1999.

[17] A. Repenning and T. Sumner, “Agentsheets: A medium for creating domain-oriented visual languages,” *Computer*, vol. 28, no. 3, pp. 17–25, 1995.

[18] K. Truong, E. Huang, and G. Abowd, “CAMP: A magnetic poetry interface for end-user programming of capture applications for the home,” *UbiComp 2004: Ubiquitous Computing*, pp. 143–160, 2004.

[19] R. Hague, P. Robinson, and A. Blackwell, “Towards ubiquitous end-user programming,” *Adjunct Proceedings of UbiComp*, pp. 169–170, 2003.

[20] A. Blackwell and R. Hague, “Autohan: An architecture for programming the home,” *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 150–157, 2001.

[21] O. Beletski, “End user mashup programming environments,” *Helsinki University of Technology. Telecommunications Software and Multimedia Laboratory T-111.5550 Seminar on Multimedia*, p. 13, 2008.

[22] M. Sabbouh, J. Higginson, S. Semy, and D. Gagne, “Web mashup scripting language,” *Proceedings of the 16th International Conference on World Wide Web*, Jan 2007.

[23] E. Maximilien, A. Ranabahu, and K. Gomadam, “An online platform for Web APIs and service mashups,” *Internet Computing, IEEE*, vol. 12, no. 5, pp. 32–43, 2008.

[24] J. Wong and J. Hong, “Marmite: End-user programming for the Web,” *CHI ’06 extended abstracts on Human factors in computing systems*, Apr 2006.

[25] J. Wong and J. Hong, “Making mashups with marmite: Towards end-user programming for the Web,” *CHI ’07: Proceedings of the SIGCHI conference on Human factors in computing systems*, Apr 2007.

[26] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. Lau, “End-user programming of mashups with vegemite,” *IUI ’09: Proceedingsc of the 13th International Conference on Intelligent User Interfaces*, Feb 2009.

[27] G. Wang, S. Yang, and Y. Han, “Mashroom: End-user mashup programming using nested tables,” *Proceedings of the 18th International Conference on World Wide Web*, pp. 861–870, Jan 2009.