Scheme in Industrial Automation

[Extended Abstract]

Marco Benelli
mbenelli@yahoo.com

ABSTRACT

This paper provides a description of a success story in using Scheme in a production environment, in the field of industrial automation. It describes the migration of an application based on legacy technologies such as Java applets and CGIs written in C, to a modern web application, through a set of smooth steps that have incrementally improved the product and the workflow. The components that have allowed this improvement are a set of declarative languages for configuration and customizations and a web framework. The Scheme implementation Gambit-C has showed itself to be well suited for the development of these components, thanks to the Scheme's nature and Gambit's own extensions.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.12 [**Software Engineering**]: Interoperability; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services

Keywords

Scheme, Gambit, functional programming, domain specific languages, web

1. INTRODUCTION

The limited resources typically found in Supervisory Control And Data Acquisition (SCADA) systems seem to be an obstacle to the development of modern web-based Human Machine Interfaces.

Since it is well known that "Hardware is Cheap, Programmers are Expensive", most manufacturers choose to upgrade the hardware and rely on most common tools such as Java, .net, Php, Python, Ruby for web development. Nevertheless, the cost is not the only drawback of a hardware upgrade but also space and power consumption are important factors. Furthermore, in mass production, even the slightest upgrade can push up the price of production.

This paper describes the evolution of a SCADA system to a more effective and flexible one. This process did not required any hardware upgrade, rather it has been ported also on cheaper hardware.

2. REQUIREMENTS

The starting point was a system that already has a web interface. The more interactive parts of the interface (data plotting and plant synopsis) were Java applets. The rest of the application was powered by CGIs (written in C) that handled data retrieval and session management.

The system was developed for an ARM-powered board with about 200 Mhz of cpu frequency and 128 MB of RAM, and used for supervising wastewater plants, but it was intended to be the base for more generic applications and to be ported onto other architectures. The resulting system is actually being used in wastewater plants, photovoltaic plants, climatic chambers and refrigerators on ARM, x86 and SH-2 powered machines.

Figure 1 shows a typical configuration of the board containing the supervisor and the Human Machine Interface, while the data acquisition is delegated to one or more Programmable Logic Controllers (PLC). This image is for illustrative purposes only, being the system used in several different applications, but it contains all the relevant elements.

One important requirement was the ease of content creation and customization by users without programming experience. Furthermore, some customers wanted the supervisor on proprietary devices, therefore, in some situations, it was not possible to install customized software. Thus, inducing the programmers to use existing storage and communication tools.

3. TOOLS

The first step was to remove interfaces based on Java applets.

These interfaces were generated with manual editing that produce an XML configuration file for each Java applet. This approach was problematic, the use of applets broke the web paradigm, and required the presence of JRE on the client machines. So it was decided to get rid of applets, and adopt a more modern AJAX interface. Seeing that the task was a source to source transformation, Lisp was the most

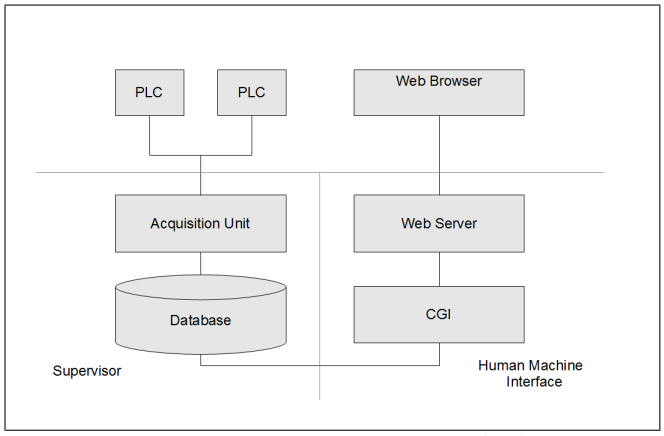

Figure 1: Components of a typical SCADA system

natural choice.

The second step was the definition of a language for building and, especially, customizing application interface.

The third step was replacing the CGI layer with a more flexible application.

Therefore a dynamic and high performance language was required. Considering possible developments, and the ability to support other platforms, portability was a desirable feature.

There are several Common Lisp and Scheme compilers that have good performance and are portable.

Scheme was chosen because of its functional features [1], valuable in XML traversing[2]; Gambit[1] was chosen because of its extensions like green threads, FFI, extended ports[3]. A lot of libraries are used: SRFIs[2], SSAX[3], irregex[4]; most of which had already been ported and optimized for Gambit. No module system like Snow[5] or Black Hole[6] were used. The namespaces and separate compilation used by Gambit satisfied our need. The macro system used is the Gambit built-in define-macro (similar to Common Lisp's defmacro). The syntax-case expander was initially used. However the overhead in size of object files was considered excessive for being used on the supervisor board.

4. DOMAIN SPECIFIC LANGUAGES

The pages most frequently used were the synopsis applets. They showed an image of the plant, with a clickable icon or text field for each physical device (pumps, switches, sensors) and a visual feedback of their state (disabled, on, off, alarm). All the configurations were read from an XML file generated by a proprietary editor, on the other hand runtime data were retrieved from the database. This configuration is illustrated in Figure 2a.

[1]http://www.iro.umontreal.ca/~gambit/
[2]http://srfi.schemers.org
[3]http://ssax.sourceforge.net
[4]http://synthcode.com/scheme/irregex/
[5]http://snow.iro.umontreal.ca/
[6]https://github.com/pereckerdal/blackhole

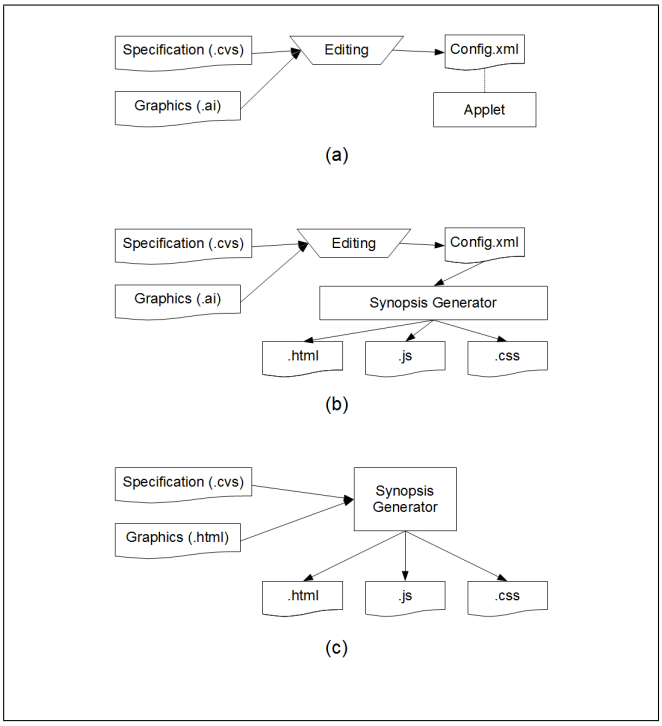

Figure 2: Evolution of workflow

A source to source compiler was written in order to remove Java applets. This compiler takes as an input the XML generated by the editor, and produces a set of HTML, CSS and JavaScripts files that create the same applet's interface. The language relies heavily on the use of SSAX-SXML. In this way, it was possible to have a new interface without any changes neither to the existing resources, or to the workflow (Figure 2b). The compiler is based on SXML: an association list maps directives (the XML tags) to functions that generate an HTML node and/or JavaScript code.

Building the XML configuration for the synopsis required the user to manually insert information about devices (type, measure units, range). This workflow was slow and prone to bug. To improve it, a new compiler was developed, that took as input an HTML map (created with common graphical editor such as Illustrator or Gimp) with the positions of the items, and a CVS file built from a spreadsheet, containing all the configurations of the devices. (Figure 2c).

This workflow proved to be effective and flexible enough to handle all the requirements that emerged in new projects (new devices, different visual feedbacks) with little or no modification to the transformer.

A similar language was developed for creating applications composed of only static HTML pages and AJAX communication. In Figure 3 there a couple of examples of usage. In the example a page with a table that shows a list of alarms is defined. The `page` directive builds the HTML page, and it is followed by the title and a specifier (`db` in this example) that gives indication of the type of page that is being built, inserting in the page initialization all the needed JavaScript code. Then `dbtable` defines a table giving it the id `alarmTable` as an HTML attribute, and builds a table using the given

sql query and fields description. Data can automatically be converted: in this example all enumerable values are showed as icons.

The second example in Figure 3 shows a variation of this language that was used in the development of web interface on a system with a supervisor that had a proprietary system for storing values. The page is a `poll` type, this means that page asks periodically for the current values of all variables (T*xxx* or T*xxx*_*x*). In the JavaScript `onload` function, there is a `setInterval` with the needed AJAX call. The following `box` directive defines a `div` element with the given id and class, to be referenced by scripts and style declarations. It should be noted that it is possible to integrate SQL and JavaScript code with the directives `sql` and `js` respectively. Embedding SQL in client code is notoriously very bad programming style, section 5 shows how this problem was solved.

Some of the directives showed in these examples produce directly html code, but there are also directives that are defined in terms of other directives, like `channel` and `toolbar` in the latter example. Composite directives were used also for whole pages, which acted as a sort of template system.

Seeing that the directives are specified in an association list, it is possible to build individual, task-specific lists that will merge when needed, for example a list for database powered widged, or for polling pages. Furthermore, since unrecognized directives are translated as identity, regular HTML tags can be used.

```
(page "Allarms" db
 (dbtable alarmTable
  (sql
   (select description tag status block mail sms)
   (from alarmlist)
   (orderby status))
  ("Description"
   "Tag"
   ("Status"
    ("Y" "led_red.png") ("N" "led_grey.png"))
   ("Blocking"
    ("Y" "tick_16.png") ("N" "void_16.png"))
   ("Mail"
    ("Y" "led_red.png") ("N" "led_grey.png"))
   ("Sms"
    ("Y" "led_red.png") ("N" "led_grey.png")))))


(page "Monitoring" poll
 (box monitor base
  (vbox
   (channel "Temperature" T001 T002 "\u00b0C")
   (channel "Humidity" T005 T006 "%")
   (hbox
    (label "Running") (led T010_0)
    (label "Auto") (led T011_2 yellow)
    (label "Alarm") (led T012_7 red)))
  (toolbar
   (btn0 "home.png"       "index.html")
   (btn1 "edit.png"       "settings.html")
   (btn6 "reset.png"      (js (blink "T100_1")))
   (btn3 "compressor.png" "pressure.html")
   (btn4 "manometer.png"  "manometer.html")
   (btn5 "leds.png"       "leds-read.html"))))
```

Figure 3: Example of DSL.

5. WEB SERVER AND APPLICATIONS

Given the constraints of the host machines, the web interface was originally written as C CGIs. This strategy was proved to be inflexible: basically the main CGI was an interface to a database that returned query results to the client as simple text. Another CGI handles authentication and sessions. The lack of a resident process forced the CGIs to continuously store and read data from the database. Some customers tried to use Python, however this rendered a poor performance. In addition to this, porting Python on some machines (ie SH-2) is not a trivial task.

The solution to this tradeoff between flexibility and performance, was found in the Klio web server, a component of the open source Klio project[7], a set of tools written in Gambit Scheme. The idea of replacing the web server occured after extended use of a simple Scheme web server for test and debug. The web server was a slightly modified version of the one present in Gambit distribution.

The Klio web server comes from completing the implementation, by adding missing features to make it compliant with version 1.1 of the HTTP protocol such as persistent connection, caching, chunked data. It also adds some common features like plain authentication, session management, and CGI support. It does not have a solid API for building application yet: the user has to write a dispatcher function in order to handle requests. For the developement we are describing here, the dispatching was done by a hash table.

[7]http://mbenelli.github.com/klio

Some utilities that simplify the retrieval of GET and POST data and format the response have been written. While the Klio Web Server supports continuation-based interaction[5], they have not been (yet) used due to the heavy use of AJAX communication between client and server. On the other hand, first-class continuations have helped interaction on some acquisition systems, that use exclusively unix signal for interprocess communications. The signal-based InterProcess Communication of this system presented some limits, and a better solution has been developed (described at end of this section), but continuations have made possible to have a working system in a short time, and the result was more solid and efficient than expected.

In Gambit's compilation model, the whole application can be compiled in a single executable file and run on platforms that do not support dynamic loading of libraries. On the other hand, the possibility of mixing compiled and interpreted code proved useful for rapid prototyping and testing. The interaction between a multithreaded Scheme code with a C library (sqlite), has also worked well, using the sqlite bindings provided by Klio tools, based on a functional interface[4]. Using FFI in Gambit is very straightforward, Figure 4 shows an example in giving Scheme access to a C function. It is also possible to build idiomatic interfaces without the need of separate C source files of glue code; for example, the function `%sqlite-open` in Figure 4 does not need a `sqlite3` pointer as argument.

With these building blocks, the interface to sqlite consists

```
(define sqlite3-open
  (c-lambda (char* sqlite3**) int "sqlite3_open"))

(define %sqlite3-open
  (c-lambda (char-string) sqlite3*
#<<C-END
  sqlite3* db;
  int res = sqlite3_open(___arg1, &db);
  ___result_voidstar = db;
C-END
))

(define (open name)
  (let ((db (%sqlite3-open name)))
    (if (zero? (sqlite3-errcode db))
        db
        (raise (sqlite3-errmsg db)))))
```

Figure 4: FFI examples

of a single function that initializes the database and returns a handler function, which works as a left-fold iterator, an example is shown in Figure 5

```
(fold/query
  (lambda (seed . cols)
    (values #t (append cols seed)))
  '()
  (sql
    (select name value)
    (from measures)))
```

Figure 5: Usage of sqlite bindings.

The old C CGI continued to work, thanks to Gambit process ports, so the new interface was tested and benchmarked against the old one. This showed better performance and reliability.

The biggest remaining problem was still the poor interprocess communication between the web application and the acquisition unit, based entirely on signals and data sharing via a database. To overcome this weakness, a new acquisition system was written in Scheme and integrated with the web application, through the implementation of standard binary protocols like /emphmodbus and /emphfetchwrite of widespread use in PLC communication. This new system has already proved itself much faster and reliable than the previous one in some benchmarks and stress tests, but it has not been used yet in a production environment.

6. CONCLUSIONS AND FUTURE WORKS

This experience has confirmed the qualities of Lisp in creating source to source compilers and Domain Specific Languages. It is both more powerful and easier to use than other transformation languages like XSLT. Furthermore, it increases re-use of existing source code. The transformer used for configuration shares a large codebase with DSLs used in pages construction. The use of DSLs has many advantages over the usual techniques adopted in mainstream web development, in finding a compromise between expressive power and simplicity. Our DSLs gives users a simple set of directives, whose documentation is tipically a short reference to their signatures. Nevertheless, letting the user to add arbitrary elements from both the underlying language (Scheme) and the target languages (HTML, CSS, and, partially, JavaScript), they do not limit their expressiv power. The only complaint raised by some user was the excessive presence of parenthesis in syntax of languages. It would not be too difficult to use an alternative syntax, maybe using Gambit's infix notation or some kind of indentation-based syntax in the spirit of SRFI-49. The fact that customers chose to not start this effort shows that the the problem has not been perceived as a show-stopper.

The experience in developing the server-side of the system was positive. In particular, Gambit Scheme has proved its qualities of simplicity, performance and reliability: it was ported without difficulties on uncommon processors, surpassing legacy C code in performance, and never showing a problem in several month of continuous running, despite the minimal quantity of test and debug that required. The speed with which a large C code-base was ported to Scheme was determinant in overcoming the initial scepticism in a fringe language like Scheme.

There is room for improvement both in the DSLs and in the web framework.

The DSLs still have some rough edges and they are oriented on some specific kind of applications. Making them more general would not be any easy task, because it would need feedback from a large number of users.

However, there is some work in progress on Klio. Besides the aforementioned work on acquisition, there is active development in adding tools for data analisys, adding new features (SSH, web sockets), and improving scalability on multicore processors, by letting its components work on individual processes.

7. REFERENCES

[1] W. Clinger. Proper tail recursion and space efficiency. *Proceedings of the 1998 ACM Conference on Programming Languages Design and Implementation*, June 1998.

[2] O. Kiselyov. A better xml parsing through functional programming. *Fourth International Symposium on Practical Aspects of Declative Languages (PADL '02)*, January 2002.

[3] M. Feeley. Gambit Scheme: Inside Out *International Lisp Conference 2010*, October 2010.

[4] O. Kiselyov. Towards the best collection api (extended abstract). *Lightweight Languages 2003 (LL3) workshop*, November 2003.

[5] C. Queinnec. The influence of browsers on evaluators or, continuation to program web servers. *ICFP 2000 - International Conference on Functional Programming*, September 2000.