The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

Lightweight Languages as Software Engineering Tools

Diomidis Spinellis
University of the Aegean
V. Guruprasad
IBM T. J. Watson Research Center

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Lightweight Languages as Software Engineering Tools

Diomidis Spinellis
University of the Aegean
83200 Karlovassi, Samos
Greece
dspin@aegean.gr

V. Guruprasad
IBM T. J. Watson Research Center
Yorktown Heights,
NY 10598, USA
prasad@watson.ibm.com

Abstract

Software subsystems can often be designed and implemented in a clear, succinct, and aesthetically pleasing way using specialized linguistic formalisms. In cases where such a formalism is incompatible with the principal language of implementation, we have devised specialized lightweight languages. Such cases include the use of repeated program code or data, the specification of complex constants, the support of a complicated development process, the implementation of systems not directly supported by the development environment, multiparadigm programming, the encapsulation of system level design, and other complex programming situations. We describe applications and subsystems that were implemented using this approach in the areas of user interface specification, software development process support, text processing, and language implementation. Finally, we analyze a number of implementation techniques for lightweight languages based on the merciless exploitation of existing language processors and tools, and the careful choice of their syntax and semantics.

1 Introduction

Software subsystems can often be designed and implemented in a clear, succinct, and aesthetically pleasing way using specialized linguistic formalisms. In cases where such a formalism is incompatible with the principal language of implementation, we have devised specialized lightweight languages. The soundness of this approach is amply demonstrated by the use of declarative languages for specialized applications [Hug90, Mos91], the attention given to very high level languages [Use94], and multiparadigm programming research [Hai86, SDE94]. The optimum formalism for implementing a subsystem is often incompatible with the language used for the rest of the system. In such cases, we propose the use of specialized lightweight languages, designed to closely match the formalism of the problem and to be easy to implement. In this way, the linguistic distance between the specification of the problem and the implementation of its solution can be minimized, resulting in cost reductions and improved project quality. In the following sections, we analyze our method's application domain, provide a number of representative case studies, outline the techniques we use, and list its advantages and associated problem areas.

2 Application Domain and Related Techniques

Lightweight languages are particularly helpful in cases where a classical implementation would require:

Repetitive sequences of code or data In many cases, one observes very similar sequences of code repeated within the program without the possibility of avoiding the repetition by the use of procedures or macros due to limitations of the programming language. In this case, the repetitive structure can be parametrically defined as a program in a specialized language. The language's compiler translates with suitable parameter substitutions the necessary parts. In [BH95], a specialized language is used to create procedures for the Ingres relational database parameterized for a given table and in [Spi93a] parameterized C code is used for the semi-automatic implementation of Haskell library functions. Section 3.3.2 describes separately compiled and interpreted filters

Specification of complex constants The constants included in a program are often the result of calculations that can be performed by a specialized language based on some basic input, or can be experimentally deduced. As an example, the size and positioning of screen objects can be calculated using

an imperative or declarative specification of the required positioning. The implementation of the language converts the screen description into code or variable initialization constants. In section 3.1.1, we describe a simple language for defining GUI property boxes.

Communication with differing subsystems When the implemented system must communicate with subsystems that are based on different programming paradigms or technologies (e.g. PLCs, Postscript printers, MIDI instruments, SQL servers) a specialized language can be used as *middleware* to bridge the mismatch between the two parts. In section 3.3.1, we describe the implementation of a functional Postscript interpreter used for the communication between a relational database and a Postscript-based video titler. In section 3.4.2, we describe another Postscript-like language developed as a processing engine for CNCs.

In addition to the above application domains, lightweight languages can be beneficially employed to support:

The software development process When developing a large system, some parts of the process may not be directly supported by the development environment. Examples are the control of additional tools not covered by the system's development environment, cross-platform configuration options, and the semi-automatic production of documentation. Using specialized languages, these processes can be automated in a simple and concise way. These languages can be compiled into input for another tool (e.g. a series of *makefiles*), or they can be directly executed (e.g. to create test vectors). In section 3.2.1 we describe a simple language for defining the distribution layout of a software product using regular expression pattern matching, and in section 3.2.2, a hand crafted language for defining a project build process.

Multiparadigm programming Often a subsystem can be implemented using a different paradigm from the main application programming paradigm (e.g. functional, logic programming, CSP-based). In this case, it is often easy to implement a little compiler or interpreter to provide exactly the features required for the realization of the specific subsystem. In section 3.4.1 we describe a rule rewrite system compiler which was *inter alia* used for the implementation of a functional and a logic programming language. In [CP85] a specialized, event-based language is described. Section 3.4.2 also describes an experimental Prolog engine implemented over a Postscript base, that allowed one to combine object oriented and logic programming paradigms.

In other cases, a system's application environment may impose unnatural restrictions on the way a user interface is implemented e.g. by requiring that parts of a given command functionality be dispersed among different event procedures, resource declarations, and icon/help text definitions. A simple language can be used to define the functionality in an organized way and automatically create the code in the format required by the application environment.

Encapsulation of System Level Design It is often the case that an application's system level design can be best expressed in a specialized language. A lightweight language can be used in this case to encapsulate the design in a compact, intuitive, and maintainable formalism.

The specialized language can be implemented as an interpreter or as a compiler. The compiler is usually implemented as code generator whose target language is the main implementation language (examples 3.1.1, 3.4.1), but it can also be implemented as a subsystem that performs the translation at runtime as in example 3.3.1. Furthermore, the language can be implemented as a built-in interpreter as in example 3.3.1 aiding the easy modification of a system's parameters at runtime. The Unix *termcap* database and the application described in section 3.3.2 are examples of implementations as data stream filters.

Building interpreters and compilers for one-time use makes it a routine skill that becomes easy to apply again and again in different ways. It also helps in developing the analytical ability for dividing and conquering problems, and for orchestrating available tools, as illustrated in [SK97] and sections 3.1.2 and 3.1.3.

3 Representative Examples

In this section we outline a number of representative systems where we utilized the described methods. The examples are divided in the areas of:

- user interface specification,
- software development process support,
- text processing,
- multiparadigm programming, and
- language implementation.

3.1 User Interface Specification

The implementation of user interfaces can benefit from the use of lightweight languages for the following reasons:

- User interfaces being close to the surface of the program receive the largest amount of modification pressure. The specification of the interface in a specialized language makes such changes easier.
- Modern user interface design guidelines require the realization of modeless environments. However, structured programming is best suited for implementing modal environments. This clash can be lifted by using a lightweight language as a bridge between the two design philosophies.
- User interfaces gather input from an ordered set of command generators such as menus, toolbars, and dialog boxes, and scatter it to various processing modules. This gather/scatter operation can be easily described using a lightweight language.

In the following sections we provide some representative examples of user interface implementations that benefited from the use of lightweight languages.

3.1.1 CAD User Interface

When implementing a large CAD program [Spi93b] we faced the difficulty of organizing and maintaining all user-interface elements of the system in an productive and coherent way. The system currently comprises:

- 30 distinct drawable entities (such as line, or text),
- 40 selectable visible layers,
- 655 user prompts,
- 457 entity properties (such as the color or width of a line),
- 103 GUI string resources,
- 130 global commands,
- 98 types of tabular data, and
- 428 entity commands.

All the above are associated with help text and therefore need to be specified in more than one language. The system implementation, maintenance, and evolution was greatly simplified by specifying the above elements and their associations using lightweight languages. Some of the languages were very simple (the specification of the user prompts involves only the definition of the prompt code and the text for each language) while others were moderately complex. In all cases however the gap between the special purpose language and the resulting C code was larger than what could be effectively bridged by the use of data driven code, C macros, and encapsulation. Table 1 contains a summary of the system parts that were specified using a lightweight language, the source code lines in that language, the compiler size (in lines of Perl [WS90] code), and the resultant C++ code. The compilation from the specialized language source code to C++ resulted in code increases ranging from 44% to 657% with the mean increase being 284%. Of the total project size of about 135,000 lines, more than 37,000 were automatically generated. A representative example of the implementation style is outlined below.

Task	Source (lines)	Compiler (lines)	Output (lines)
GUI Strings	468	67	771
User Messages	2490	65	4345
Layer Control	91	244	711
Table Contents	266	618	1901
Toolbars	373	187	1568
Menus	42	81	401
Serialization	1174	297	6191
Commands	1214	248	7367
Properties	1093	839	14632
Total	7211	2646	37887

Table 1: CAD user interface specification: source, compiler, and generated code

One of the specifications for the system called for a property dialog box similar to the one found in the Delphi, MS-Access, and Visual Basic programming environments. The system supports 457 properties divided into 30 groups. Every property was given a property type such as number, angle, color, menu selection, yes/no, dialog box, and file name. We then formulated the properties in the language as illustrated in the example in Figure 1. A compiler translates the property definitions into three C code files. One of the files contains the variable and procedure definitions, the other, the procedures for initializing the supporting data structures, and the third one contains code for displaying the properties.

The compiler consists of 839 lines of Perl code and produces 14632 lines of C code. Before the compilation, the source code is passed through the C language preprocessor allowing the use of macro instructions at a minimal cost.

The description file is compiled into C in a single pass by writing the declarations and the code definitions into two distinct files; the definition file includes the declara-

```
// Standardized selections
#define ANGLE 0;45;90;135;180;225;270;315
#define TSIZE 6;8;10;12;14;16;20;24

// Text object property selection
#ifdef TEXT
//Type Caption Variable    Range  Fmt  Sel
menu  :Text   :prop_text
double:Size   :cb->size   :0 :100:%.3f:TSIZE
angle :Angle  :cb->theta  :0 :90 :    :ANGLE
bool  :Enhance:cb->enhance:      :
color :Color  :cb->color  :      :
separator
selres:Font   :cb->font   :3     :font_dir
sel   :Allign :cb->allign :Left;Right;Center
dialog:More...:CTextDialog:textdlg
#endif
```

Figure 1: Specification of a property dialog box

tion file using the C #include mechanism. In implementing the above example we made extensive use of Perl's variable substitution facility for creating customized and efficient C code.

3.1.2 Voice Shell

In a transportation management application, a voice shell (*vsh*) was developed to interface incoming telephone calls to an online database, prompted by the highly repetitive nature of code in a previously existing application. *Vsh* incorporated a *termcap*-like stack, arithmetic, voice prompts, keystroke data entry, shell escapes, and submenu invocation control, compactly representing the logic of a complete voice response system comprising about 75 speech files and 50 database access scripts within a 24x80 ASCII screen with ample space for comments. Not only was development time reduced for similar applications, but, in addition, the database accesses were simplified to short lightweight scripts. To compare, prior versions of the application would each contain 10000 or more lines of C with deeply nested "if"s, and would be generally inflexible and difficult to debug.

3.1.3 Web-based System

Reduction to lightweight languages and scripts is again a key aspect of the Papyrus online conference paper review system [Gur96]. The system comprises mostly *ksh* scripts; entire subsystems can be easily customized, thrown away or replaced. The flexibility was critical in the evolution of the package with the live requirements of an actual conference (PACT'96). The system has been or is in the process of being adopted for at least two other international conferences this year. Language components include review questionnaires parsed and evaluated using *awk* and author response form letters generated using *nroff* to format the email text.

3.2 Software Development Process Support

The software development process is a less mature field than programming in the small. The requirements, organization techniques, procedures, and development tools vary widely between organizations and projects. Lightweight languages can provide support for the above by automating repetitive work, organizing complicated tasks such as configuration control, and forcing ill-defined processes onto a formal specification vehicle. The following paragraphs outline two representative examples in the area of software development process support.

3.2.1 File Disk Layout

In a large application that we developed we had to specify the ordering of 530 files in the installation disks as well as the associated parameters. Some of the files were needed only during the setup stage and should be placed in the first disk, some of them should, additionally, not be compressed. Other files had to be installed into specific system directories while others had to be installed only as an installation option. This layout is typically specified by a GUI tool provided as part of the Microsoft Windows software development kit. The tool then distributes the files to disks and creates the installation driver program. In our case, the GUI tool could not handle the number of files needed by our application, and, in addition, had only limited grouping capabilities. We thus implemented a little language based on regular expressions that defines file sets and their installation specifications together with some global settings. Part of this specification can be seen in Figure 2.

The specification file is then translated into a file compatible with the one created by a GUI tool. In our case, the regular expression-based specification file is 66 lines long and creates a 560 line specification. The short length of our specification makes it comprehensible, readable, and ammendable to processing by other tools.

A 55 line Perl program handles the translation, identifies simple mistakes, and prints warnings for the files that fall into the default, the most general specification case, so that the user can verify that the files are given valid installation specifications.

```
# Anchor =
/distrib/
# The following are copied verbatim
=SRC = \distrib
=WRITEABLE = 1
=DISKLABEL = Application Disk 1
# Installation files (do not compress)
>DISK1,!COMPRESS,!VERSION,!READONLY,
    !DECOMPRESS,SECTION=Setup,.
+setup.exe
+setup.lst
# Library files
>ANYDISK,COMPRESS,!VERSION,!READONLY,
    DECOMPRESS,SECTION=LibFiles,.
+lib/*
# All other files (application files)
>ANYDISK,COMPRESS,!VERSION,!READONLY,
    DECOMPRESS,SECTION=AppFiles,.
+*.exe
+helplib.dll
+*
```

Figure 2: Installation file layout specification

3.2.2 Generation of Makefiles

A very simple rule-based interpreter called *archmake* provided such effective documentation and control of the build process for a 1986 project involving a few hundred source files, that it was quickly adopted as the self-documenting in-house build tool for several other multi-programmer projects before CASE tools became available. Unlike *make* [Fel79] and *imake*, the interpreter had no built-in expertise, but provided a concise and easy-to-read representation of the target object name, the component source list, the dependency generation command script, and the unit compilation script, in the following form:

```
%S description of step
# list component files to work on
%L a.c b.c c.c ...
%D mkdepend.sh %s
...
%M $(CC) -c $@ $*.c
...
```

3.3 Text Processing

As processor speeds increase, text is increasingly used as a common communication format between different subsystem parts in a quest for portability and maintainability. However, most compiled programming languages lack functionality for dealing efficiently and intuitively with text strings. The design of a small lightweight language to be embedded in a system can be used to ameliorate this deficiency by providing an efficient implementation of the facilities needed for using the specific textual interface. Two representative examples are described in the following paragraphs.

3.3.1 Video Typesetting

A statistical data relational database program we developed had to interface to a specialized video typesetting device. The device was controlled using Postscript [Inc85]. Structuring all the resulting code around Postscript was not feasible as Postscript is a relatively low-level language. Embedding large blocks of Postscript into the program was inelegant. We therefore defined a Postscript variant as a lightweight language that allowed for the embedding of meta-variable names. These names (all starting with the $ sign) were substituted at the runtime interpretation phase with the SQL query results. In order to allow for the definition of tabular data with variables of the same name the semantics of the language mimic *linear logic* in allowing each variable substitution to happen exactly once. An external loop within the code thus calls the interpreter multiple times for the same code body, each time setting the variable names to the values of the particular table row. After the implementation, all Postscript code that was embedded within the rest of the program was easily moved into external files. When a particular page needs to be displayed the file is loaded, interpreted, and downloaded into the Postscript typesetter. Most of the Postscript code of the deployed release was generated by experienced artists using specialized graphic design tools.

3.3.2 Rendering Indian languages

The Prototext language [Gur88] was designed to provide runtime filters for working with Indian language scripts using *unmodified* text editors and word processors. Indian scripts comprise a near-isomorphic family of phonetic scripts with strict rules for the graphic composition of syllabic glyphs from vowel and consonant components, and are precursors to the simpler syllabic alphabets of the Far East. The interpreter allowed the repetitive bitmap operations and the script-specific composition rules to be efficiently and compactly implemented independently of text processing code. It also facilitated the development of the fonts and composition rules on ASCII terminals before graphic editing facilities became available, and provided slightly higher performance than the not-so-optimized C code performing the same function.

3.4 Multiparadigm Programming and Language Implementation

Either in response to a genuine need, or through the occupation of individuals with domain specific knowledge, the area of programming language implementation has traditionally been a hotbed for lightweight languages. The increasing use of multiparadigm and mixed-language programming, and the possible adoption of lightweight language tools that we advocate, increase the requirements for an efficient process of language implementation. We believe that the proliferation of languages used for implementing languages stems from the need to specify formal transformation rules for a wide variety of domains that occur in a language realization. The use of a special purpose (for mature tasks such as parsing) or a lightweight language can aid the specification — and a subsequent implementation — of tokens, grammars, type systems, tree transformations, optimizations, and machine code generators. Below we outline three representative examples from our own work.

3.4.1 Rule-based Programming

We designed and implemented *term* as a lightweight language for the realization of the *blueprint* multiparadigm environment [SDE94]. For that environment we needed to implement a functional and a logic programming language. Such languages are easily implemented using declarative rules as *meta-interpreters* in their own language [SS86, p. 150] [FH88, pp. 193–195]. This practice is unfortunately associated with serious bootstrapping and performance issues. Based on the intuition that with only a moderate increase in code size the above languages can be implemented in a language that does not support important features of functional and logic programming such as deep unification, input/output parameters, backtracking, and higher order functions, we defined *term* as a simple tuple pattern matching, rule-rewrite system. *Term* is implemented in 2300 lines of *term*, yacc, and lex as a compiler from *term* to C. It was bootstrapped using a Prolog interpreter since the syntax of *term* is very similar to that of Prolog, lacking however many of Prolog's logical features. The use of *term* as an implementation vehicle proved to be a good choice, since we were then able to implement the functional and the logic languages in just 1300 lines of structured *term* code that was then compiled into efficient C code.

3.4.2 CNC extensions to Postscript

Faced with developing a complete APT (Automatically Programmed Tools) language system for CNC applications [Kra86, CM89], we found it expedient to partition the problem into a simple 22-rule Yacc-based parser front-end to translate APT to a lightweight, Postscript-like back-end language, *4th*, with minimal syntax checking. The semantic validation and *N*-D path geometry was developed directly in *4th*, which even had a *goto* to match the Fortran-ish flavor of the APT source language. The path profiling code was later reduced to a C-based extension to the interpreter, which remained as the base engine. The *4th* core was inspired by good microprocessor design: it had a control and status word, provided traps, single-stepping and call tracing, and allowed object-oriented programming support using software traps in *4th* itself. The design achieved portability, scalability, capability for backending to CAD tools, and extendibility since builtin operator table could be easily added to. This approach allowed the reduction of our profiling algorithm to three operators, implemented in 1200 lines of C and using about 1000 lines of supporting code in *4th* for full 2-1/2 D APT geometry.

An experimental Prolog extension was also built using the dictionary objects as independent "knowledge domains", unlike most implementations of Prolog, which provide only a default namespace for the functors, and provided *4th* escapes like:

```
% 4th.logic init file
{
% escape to perform inits in true 4th
(/usr/lib/4th.init) ldsrc
% push to logic domain stack
/lathe_rules load beginlogic
}
% Note-- :n means n-th from top of stack.
=(X,X) :- { :0 :1 unify }.
```

The logic programming feature could be used, for instance, to integrate tool-specific reasoning in factory automation.

3.4.3 Haskell Implementation

Haskell [HF92] is a complete general purpose purely functional programming language. When building a Haskell system [Spi93a] we utilized a number of lightweight languages for implementing:

- an efficient character classification interface based on regular expressions used for lexical analysis,
- routines for reversing, printing, and copying arbitrary parse trees,
- the language library functions in a high level form, and
- a generic machine description interface.

Of the project's total of 17,268 lines 3,610 were automatically generated. Table 2 summarizes the system parts that

Task	Source (lines)	Compiler (lines)	Output (lines)
Character classification	44	87	332
Tree inversion	527	223	441
Tree printing	527	244	1175
Tree copy	527	206	507
Library functions	178	131	410
Machine description	604	274	745
Total	1353	1165	3610

Table 2: Haskell implementation: source, compiler, and generated code

were specified using a lightweight language, the source code lines in that language, the compiler size (lines of Perl code), and the resultant C code. It is important to note that the compilers that created the tree operation routines all used the same parse tree definition C header file as source input. In this way any changes to the parse tree were automatically reflected by new tree handling routines.

4 Implementation Techniques

Lightweight languages are by their nature ephemeral and limited in scope. The effort expended in implementing them should be less than the effort required to build the system without their use. In the following sections we will outline some implementation techniques that we have successfully used to implement the languages we described in a robust yet economical way.

Use existing tools

Language implementation is a mature field in theory and technology and all relevant knowledge and tools should be capitalized when realizing a special-purpose language. The language implementation problems, solutions, and tools are taught at most computer science courses and can be put to good use when building a system in this way. Therefore, the implementation of a specially-built, lightweight language can be a valid, realistic, and cost-effective proposition.

Many simple interpreters can be implemented using text processing tools such as awk [AKW88] and Perl [WS90]. This implementation avenue can be particularly effective if the language has been designed by following the guidelines outlined in the following four paragraphs. Interpreters and compilers for more complicated languages can always be modeled using the lex/yacc [JL87] model; often performing the translation or code generation during parsing without a need to create an intermediate syntax tree.

Combine language processors

Many of the lightweight languages that we have implemented are simply preprocessors that emit another high level language. This implementation strategy is used by *yacc*, *lex*, and the *cfront* C++ implementation. In addition, the semantic richness of the lightweight language can be increased by judicious combination of a number of language processors. In many cases we pass our lightweight language source code through the C preprocessor thus adding to the language file inclusion, macro definitions, and conditional compilation facilities.

Use the features of the target language

The compilation of lightweight languages can be made easier by translating many statements or expressions directly into statements or expressions of the target language. The compiler input language can allow for parts that are not processed by the compiler, but are passed – perhaps after some minor variable name substitutions – directly to the compiled code. Languages that are compiled into C can use the `#line` preprocessor instruction to make the C compiler errors refer to the source language and not the target language. Furthermore, one pass compilation of the lightweight language can often be achieved by directing the compiled code into two files (e.g. declarations and definitions), one of which includes the other.

Use the features of the implementation tool

One other way to increase the capabilities of the lightweight language with little implementation cost is to utilize the power of the implementation tool. As an example, some tools have meta-linguistic capabilities allowing the interpretation of their own language. In this case the lightweight language source code can include expressions or statements written in the implementation language of the interpreter. In one of the languages we implemented we allowed the inclusion of full Perl expressions since these can be easily evaluated using Perl's *eval* function.

Imitate implementation tool features

Simple but robust interfaces can be easily built even with *ksh* [Kor94]. We have used simple *ksh* loops for everything ranging from compilation scaffolds and simple Web servers to entire online services (sections 3.1.2 and 3.1.3). Including simple features like shell escapes and I/O redirection is not only convenient for Unix users, but also helps in scripting and logging production runs.

Use simple syntax and provide lexical hints

To successfully use the techniques described above it is important to design the lightweight language in such a way as to make it compatible in syntax, semantics, and form with the target language, the interpreter language, or the C preprocessor.

By keeping the lightweight language small and limited in scope it should be possible for a single person to design and implement it. By a judicious selection of syntax and semantics the language can be implemented without complicated processing. Many of the languages we have designed are line and not free-form based. They can thus be compiled one line at a time, sometimes by a simple application of pattern matching and regular expressions. Line comments are also easy to remove without complicated loops and the semantic problems associated with nested block comments. An even easier to compile form of language syntax is table-based (as the example in section 3.1.1) and can thus be easily processed by field-based tools such as *awk*. The processing of variables can often be simplified be prefixing them with a special character such as $.

Make source files self-documenting

Systems we have implemented made use of as many as 13 different languages. One cannot expect a person maintaining the code to be familiar with all of them. For this reason we try to make every source file self-documenting. We try to make the language similar to well-known languages and keep its syntax and semantics simple and intuitive. A comment at the beginning of each source file of no more than a dozen lines should be all that is needed to describe the language and its use. If this is not possible, then something may be wrong with the language design.

Reprocess existing source code

The source code of a project, when written following certain conventions, can be used as as an input language for a tool that will provide additional functionality. We have thus successfully parsed header files to create a localization message database, tree processing functions (section 3.4.3), and, a variable load / save facility. In another application we have embedded error messages, explanation text, and recovery actions as comments in the source code that was processed in a later stage to create an external error message database.

5 Problems

Project architects contemplating the use of lightweight languages should carefully weigh the advantages outlined in the previous sections against a number of potential problems.

The software process covering the use of lightweight languages is not yet mature. Issues of lightweight language design in the areas of granularity, usability, interfacing, and architecture need to be examined and evaluated. The potential scalability limits of this approach should also be taken into consideration. We have used lightweight languages in projects composed of up to 200K lines of code without experiencing significant problems. Significantly larger projects may uncover problems in the areas of namespace pollution, tool portability, performance, group coordination, and resource management.

Furthermore, the developers and users of lightweight languages have to support their language on their own. Tools such as debuggers, metric analyzers, profilers, class browsers, and context-sensitive editors will be not be available. Users will have to do without them, handcraft their own, or resort to the lower level facilities provided by the underlying language. Other resources commonly available for mainstream languages such as trained developers, courses, libraries, books, and external consultants will also be absent. In addition, developers may need to reinvent techniques used for formal program validation and analysis, and re-establish metrics associated with the development process.

6 Related Work

The problem of choosing an application's implementation language forms part of language design research [Wir74, Wex76, Hoa83]. In this section we examine some representative specialized languages that justify our approach. The implementation of a number of specialized languages can be found in [AKW88] and [Ben88, pp. 83–100]. A data driven approach to structure programs is recommended in [KP78, p. 67]: "let the data structure the program." An example of a language with a narrow, specialized application domain is the one described in [CP85]. Specialized languages and tools for compiler construction are given in [JL87, Fra89, Spi93a]. Furthermore, specialized language dialects can be implemented using the system described by [CHP88].

The Unix operating system follows the tradition of using a specialized language for the definition of an application's startup status. Some of these languages are sufficiently advanced so that part of an application can often be implemented in them. As an example the *sendmail* mail transfer agent only implements a simple translation engine configured by a special startup file. All rules for routing and rewriting message headers are defined in that external configuration file (the infamous *sendmail.cf* file). Simpler yet useful examples of spe-

cialized languages in the Unix environment are the system state initialization file *inittab*, the terminal capability descriptions in *termcap*, the process schedule specification *crontab*, and the Internet daemon master switchboard *inetd*. The plethora of command initialization files for tools such as the X-Window system (*.xinitrc*), the ex editor (*.exrc*), the window manager (*.twmrc*), and the mail user agent (*.mailrc*) prompted the development of TCL [Ous94], an embedable, interpreted, string processing language aiming to provide a single consistent tool programming interface across all different tools. Finally, the *troff* [Oss82] family of Unix-based text processing tools, builds on small specialized languages for typesetting equations (*eqn* [KC74]), tables (*tbl* [Les82]), pictures (*pic* [Ker84]), and graphs (*grap* [BK86]).

Our approach builds on the ideas mentioned above by promoting the use of specialized lightweight languages as an integral part of the software development process.

7 Conclusions

The use of lightweight languages as software engineering tools bridges the semantic gap between the specification and the implementation, can offer economies of scale when implementing repetitive concepts, and can result in code that is readable, compact, easy to maintain, and concisely documents the overall structure of the application.

However, the proliferation of different languages within a project can contribute problems related to the poorly understood process, lack of tools, and scarcity of related resources. All these elements need further examination and research. We strongly believe that this research will extend the applicability of our approach and amplify the advantages brought by the use of lightweight languages in the software development process.

References

[AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[Ben88] Jon Louis Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1988.

[BH95] Olin Bray and Michael M. Hess. Reengineering a configuration-management system. *IEEE Software*, 12(1):55–63, January 1995.

[BK86] Jon Louis Bentley and Brian W. Kernighan. GRAP — a language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, August 1986.

[CHP88] James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings IEEE 1988 International Conference on Computer Languages*, Miami, USA, October 1988. IEEE Computing Society.

[CM89] Chao-Hwa Chang and Michael A. Melkanoff. *NC Machine Programming and Software Design*. Prentice-Hall, 1989.

[CP85] Luca Cardelli and Rob Pike. Squeak: a language for communicating with mice. *Computer Graphics*, 19(3):199–204, July 1985. SIGGRAPH '85 Conference Proceedings, July 22–26, San Francisco, California.

[Fel79] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, 1979.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[Fra89] Christopher W. Fraser. A language for writing code generators. *ACM SIGPLAN Notices*, 24(7):238–245, July 1989.

[Gur88] V. Guruprasad. Prototext: Universal text drivers. In *Summer 1988 Usenix Conference*, pages 331–338, San Francisco, CA, USA, June 1988. Usenix Association.

[Gur96] V. Guruprasad. Papyrus: an online paper reviewing system. http://www.pact96.ibm.com, 1996.

[Hai86] Brent Hailpern. Multiparadigm research: A survey of nine projects. *IEEE Software*, 3(1):70–77, January 1986.

[HF92] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Haskell Special Issue.

[Hoa83] C. A. R. Hoare. Hints on programming language design. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, pages 31–40. Computer Science Press, 1983. Reprinted from Sigact/Sigplan Symposium on Principles of Programming Languages, October 1973.

[Hug90] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17–42. Addison-Wesley, 1990. Also appeared in the April 1989 issue of The Computer Journal.

[Inc85] Adobe Systems Incorporated. *Postscript Language Reference Manual*. Addison-Wesley, 1985.

[JL87] Stephen C. Johnson and Michael E. Lesk. Language development tools. *Bell System Technical Journal*, 56(6):2155–2176, July-August 1987.

[KC74] Brian W. Kernighan and L. L. Cherry. A system for typesetting mathematics. Computer Science Technical Report 17, Bell Laboratories, Murray Hill, NJ, USA, May 1974. Revised April 1977.

[Ker84] Brian W. Kernighan. PIC — a graphics language for typesetting: Revised user manual. Computer Science Technical Report 116, Bell Laboratories, Murray Hill, NJ, USA, December 1984.

[Kor94] David G. Korn. Ksh - an extensible high level language. In *Very High Level Languages Symposium (VHLL)*, pages 129–146, Santa Fe, NM, USA, October 1994. Usenix Association.

[KP78] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, second edition, 1978.

[Kra86] Irving Kral. *Numerical Control Programming in APT*. Prentice-Hall, 1986.

[Les82] Michael E. Lesk. TBL — a program to format tables. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 157–174. Holt, Rinehart and Winston, seventh edition, 1982.

[Mos91] Chris Moss. Commercial applications of large Prolog knowledge bases. In H. Boley and M. M. Richter, editors, *Processing Declarative Knowledge: International Workshop PDK '91 Proceedings*, pages 32–40, Kaiserslautern, Germany, July 1991. Springer-Verlag. Lecture Notes in Computer Science 567.

[Oss82] J. F. Ossanna. NROFF/TROFF user's manual. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 196–229. Holt, Rinehart and Winston, seventh edition, 1982.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[SDE94] Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach. Language and architecture paradigms as object classes: A unified approach towards multiparadigm programming. In Jürg Gutknecht, editor, *Programming Languages and System Architectures International Conference*, pages 191–207, Zurich, Switzerland, March 1994. Springer-Verlag. Lecture Notes in Computer Science 782.

[SK97] Diomidis Spinellis and Rob Kolstad. A conversation about Perl and the shell: Choosing the implementation vehicle. *;login:*, 22(3):25–31, June 1997.

[Spi93a] Diomidis Spinellis. Implementing Haskell: Language implementation as a tool building exercise. *Structured Programming*, 14:37–48, 1993.

[Spi93b] Diomidis Spinellis. Tekton: A program for the composition, design, and three-dimensional view of architectural subjects. In *4th Panhellenic Informatics Conference*, volume I, pages 361–372, Patras, Greece, December 1993. Greek Computer Society. In Greek.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Use94] Usenix Association. *Very High Level Languages Workshop (VHLL)*, Santa Fe, Mexico, October 1994. Usenix Association.

[Wex76] Richard L. Wexelblat. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 331–336, San Fransisco, CA, USA, October 1976. IEEE Computer Society Press.

[Wir74] Niklaus Wirth. On the design of programming languages. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of IFIP Congress 74*, pages 386–393, Stockholm, Sweden, August 1974. International Federation for Information Processing, North-Holland Publishing Company.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, 1990.