The Simplicity and Safety of the Language for End System Services (LESS)

Xiaotao Wu and Henning Schulzrinne
Department of Computer Science
Columbia University
xiaotaow,hgs@cs.columbia.edu

July 20, 2004

Abstract

This paper analyzes the simplicity and safety of the Language for End System Services (LESS).

1 Introduction

The Language for End System Services (LESS) [5] is designed for programming communication services in end systems and used by end users without programming experience. The goal of the language is to allow end users to program their own communication services with little training in a graphical service creation environment. To achieve the goal, the language must represent a high-level abstraction of communication behaviors. The elements in the language must have semantic meanings. The language has to be simple and safe. In this paper, we analyze how the language provides simplicity, safety, and high-level abstraction of communication services.

Section 2 describes the high-level abstraction LESS uses to represent communication services. Section 3 investigate why LESS offers simplicity. Section 4 investigate how LESS handles programming safety. Section 5 concludes the paper.

2 High level abstraction

Figure 1 shows the service model of LESS. A call decision making process in LESS consists of three steps: invoking triggers, branching call decisions based on switches, and performing communication actions.

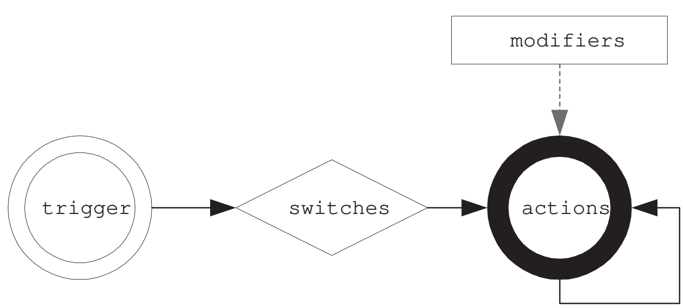

Figure 1: Call decision making process in LESS

A service starts when a trigger invoked. A trigger can be an incoming call, an outgoing call, or a timer event. New triggers, such as event notification, can be introduced by defining LESS extensions. In a LESS script, switches check the status of the trigger and its context. For example, an `address-switch` may check the caller and the callee's addresses, and based on the addresses to make call decisions. Call decisions are executed by performing communication actions, such as `accept`, `reject`, or `redirect` a call. The modifiers are used to provide action arguments. For example, the `location-modifier` element indicates the URI to `redirect` a call to. Additional actions may get performed based on the results of their previous actions. The abstraction simulates people's natural thinking for call decision making and is easy for users to understand and learn.

The high-level abstraction implies a tree-like structure for call decision making as shown in Figure 2.

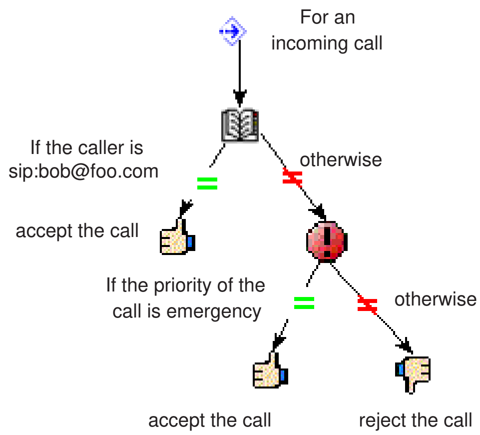

Figure 2: Tree-like structure for call decision making

The tree-like structure makes LESS easy to analyze and safe for service programming. We will detail the analyzability and safety of LESS in Section 3 and Section 4.

3 Simplicity

"The fewer concepts to understand in a language the better" [4], though, in some cases two smaller concepts might be simpler and more flexible than one more powerful but complicated concept. For example, separating commonly used action arguments, such as people's contact locations, as modifiers is better than putting the arguments into every action.

"Simplicity enters in four disguises: uniformity (rules are few and simple), generality (a small number of general functions provide as special cases of more specialized functions), familiarity (familiar symbols and usages are adopted whenever possible), and brevity (economy of expression is sought)" [1]. For LESS, we consider analyzability is more important than brevity because LESS is an XML-based language designed for communication services and has people without programming experience as its target users. In communication service creation, users may encounter feature interaction [6] problems, how to analyze and solve feature interactions is essential for users to create usable services. We discuss the generality, uniformity, familiarity, and analyzability of LESS below.

3.1 Generality

The generality requires a language to have a small number of general elements. The special elements of the language can be introduced by specializing the general elements. As illustrated in Section 2, LESS has only four kinds of elements, namely triggers, switches, actions, and modifiers. Every new element defined in LESS extensions must fall in one kind of the elements. In another words, in LESS's XML schema, LESS defines four basic abstract elements, namely 'trigger', 'switch', 'action', and 'modifier'. In the XML schema of LESS extensions, every element must be a `substitutionGroup` of one of the four abstract elements.

3.2 Uniformity

The uniformity requires a language with few and simple rules. There are only four rules in LESS. The rules can be summarized below:

Trigger rule: A LESS script gets invoked if and only if one of its triggers get matched. A trigger must be the root of a decision tree and can appear no more than once in a script.

Switch rule: Only switches can branch call decisions. One switch have no more than two branches. A switch must be a child of a trigger or another switch in a decision tree.

Action rule: Only actions can change call status and call context and only actions can be LESS decision tree leaves. There can be subsequent actions after one action. No switches or triggers can follow an action.

Modifier rule: A modifier can only be used as the parent element of actions to enforce the actions.

3.3 Familiarity

The familiarity requires a language not to violate common use of notation. LESS is an XML-based language and uses the datatypes defined in XML schema. The elements defined in LESS all have semantic meaning so it is easy for users to understand and remember.

3.4 Analyzability

Another aspect of language simplicity is how easy to analyze a program written by the language. For LESS, we are more interested in how LESS handling feature interaction problems in communication services. As discussed in our technical report "Feature Interactions in Internet Telephony End Systems" [6], we can handle feature interaction problems among LESS scripts by the action conflict tables and the tree merging algorithm defined in the technical report. The process is straight-forward, and can not only detect feature interaction detection, but also help to solve the interactions.

4 Safety

Since LESS will be used by people without programming experience, we may expect errors that seems naive by experienced programmers. We should also expect that people may use third-party created malicious service scripts. The language design should help people to prevent errors and protect people from malicious scripts. The error prevention machanisms for LESS fall into two categories, one is to put restrictions on the language itself, the other is to put restrictions on LESS engines. We are more interested in the restrictions on the language itself since that is directly related to the language design. The restrictions on LESS engines may cause service portability problem so we should try to prevent errors in the language design whenever possible.

4.1 Type safety

Type checking is a very effective way in catching programming errors, from the trivial errors, such as misspelled identifiers, to the fairly deep errors, such as violations of data structure invariants.

LESS is an XML-based language and uses XML schema [2] to define its elements. "The strong typing mechanism in XML Schema, along with the large set of intrinsic types and the ability to create user-defined types, provides for a high level of type safety in instance documents. This feature can be used to express more strict data type constraints, such as those of attribute values, when using XML Schema for validation." [2]

In LESS, there are no user defined variables so only static type checking in needed. There are many advantages to have a statically type-checked language. Static type checking can provide earlier, and usually more accurate information on programmer errors. It can eliminate the need for run-time checks, which may slow program execution. Statically type-checked language may be less expressive than dynamically type-checked language, however, since LESS is not designed to handle all communication services and targets to end users, safety is more important than expressiveness.

4.2 Control flow safety

LESS has a tree-like structure for call decision making. LESS does not allow a tree node to back-reference to its ancestors or itself. This means there is no loop or recursion in LESS scripts, and will exclude the possibility of non-terminating or non-decidable LESS scripts. However, LESS still provides some mechanisms for repeated events handling. The `timer` trigger can get invoked periodically. The `lookup` modifier implies that follow-up actions are applied to every location in the result set.

Run-time feature interactions may confuse users. The LESS trigger rule ensures that a specific trigger can appear no more than once in a LESS script thus helps to avoid run-time feature interactions in the LESS script. The LESS service creation environment should help users to merge multiple service scripts into one as illustrated in the technical report on feature interaction handling in LESS [6].

4.3 Memory access

A language can be described as unsafe in that the language allows some means to access memory directly. In LESS, there are no pointers, no direct memory accesses, and even no user-defined variables. LESS has maximum string length defined in its XML schema for every element with string type to prevent buffer overflow attack. If a user put a very long string in a service script, the service script will be considered invalid when checking against the LESS XML schema.

4.4 LESS engine safety

LESS engine developers can do several things to enhance the safety of LESS scripts. The developers must make sure that every trigger has default behaviors defined. If a trigger gets invoked in a LESS engine, but there is no actions defined for the matching context, the LESS engine must perform the default actions for that trigger. This ensures every LESS script is decidable.

LESS is different from CPL [3] when dealing with user trustiness. CPL is designed for running on signaling servers, such as SIP proxy servers. The safety consideration for CPL ensures that semi-trusted users cannot create malicious or incompetent service scripts to interrupt other users' services, including crashing the server, revealing security-sensitive information, and causing denial of service. While LESS is used on users' end devices, usually, it does not need to handle the interference of other users. LESS engine need to ensure safe resource usages, such as CPU usage. It needs to define how deep a decision tree can be, what is the minimum interval for `timer` trigger, and whether it can control multimedia devices.

4.5 Using third-party created or auto-generated service scripts

Sometimes, end users may not be aware of a new service or they may not know how to create a service, the service scripts auto-generated by feature learning, or created by third-parties, may bring great conveniences to users. However, end users have to check the safety of the third-party created or auto-generated scripts, for example, to make sure that the scripts will not crash their systems or forward their calls to unwanted parties.

The safety consideration discussed in previous sections can be applied to third-party or auto-generated service scripts. In addition, the simplicity and the tree-like structure of LESS make it viable to convert any valid LESS scripts into graphical representation of decision trees. This is a big advantage of LESS for end users to do safety checking. Users do not have to check LESS programs line by line, instead, they can watch the graphical representation and easily find out what actions a service script will perform.

5 Conclusion

From the analysis, we conclude that LESS is simple and safe and it is well-suited for people without programming experience to perform end system service creation.. However, the simplicity and safety of LESS may make it impossible or very difficult to handle some complicated services. Intrinsically, LESS is not Turing complete so it cannot handle all the services. The lack of loop and user defined variables make the language only good for a limited kind of services. We are working on a separate technical report to enumerate what services can be handled by LESS. The initial analysis of LESS services shows that, though the functionality of LESS is limited due to its simplicity and safety concern, it can still handle a wide range of commonly used services.

References

[1] A. D. Falkoff and K. E. Iverson. The design of apl. *SIGAPL APL Quote Quad*, 6(1):5–14, 1975.

[2] David C. Fallside. XML schema part 0: Primer. W3C Candidate Recommendation CR-xmlschema-0-20001024, World Wide Web Consortium (W3C), October 2000. Available at http://www.w3.org/TR/xmlschema-0/.

[3] Jonathan Lennox, Xiaotao Wu, and Henning Schulzrinne. CPL: a language for user control of Internet telephony services. Internet draft, Internet Engineering Task Force, August 2003. Work in progress.

[4] Ryan Stansifer. *Study of Programming Languages*, volume 1. Prentice Hall, Florida Institute of Technology, 1994.

[5] Xiaotao Wu and Henning Schulzrinne. Programmable end system services using SIP. In *Conference Record of the International Conference on Communications (ICC)*, May 2003.

[6] Xiaotao Wu and Henning Schulzrinne. Feature interactions in Internet telephony end systems. Technical report, Department of Computer Science, Columbia University, January 2004.