SmART: An Application Reconfiguration Framework

Hervé Paulino, João André Martins, João Lourenço, and Nuno Duro

Abstract SmART (Smart Application Reconfiguration Tool) is a framework for the automatic configuration of systems and applications. The tool implements an application configuration workflow that resorts to the similarities between configuration files (i.e., patterns such as parameters, comments and blocks) to allow a syntax independent manipulation and transformation of system and application configuration files. Without compromising its generality, SmART targets virtualized IT infrastructures, configuring virtual appliances and its applications. SmART reduces the time required to (re)configure a set of applications by automating time-consuming steps of the process, independently of the nature of the application to be configured. Industrial experimentation and utilization of SmART show that the framework is able to correctly transform a large amount of configuration files into a generic syntax and back to their original syntax. They also show that the elapsed time in that process is adequate to what would be expected of an interactive tool. SmART is currently being integrated into the VIRTU bundle, whose trial version is available for download from the project's web page.

Key words: Automatic configuration, Virtualization, Virtual appliance

Hervé Paulino · João Lourenço
CITI / Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal. e-mail: {herve,Joao.Lourenco}@di.fct.unl.pt

João André Martins
Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal. e-mail: citanul86@gmail.com

Nuno Duro
Evolve Space Solutions, Centro de Incubação e Desenvolvimento, Lispolis, Estrada Paço do Lumiar, Lote 1, 1600-546 Lisboa, Portugal. e-mail: nuno.duro@evolve.pt

1 Introduction

Virtualization techniques and technologies have been around for quite some time in mainframe environments. However, only more recently, with the advent of low cost multi-core processors with support for hardware virtualization, and a wider operating system support, the use of virtualization is spreading at a fast pace.

This widespread use raises several issues on the automation of the configuration and deployment of applications to be executed in Virtual Appliances (VA – customized virtual machine image). Anticipating some of these issues, a consortium including both Industry and Academic/Research partners is currently tackling the automation of the configuration and deployment of applications in VAs in the scope of the VIRTU project [3]. The consortium involves Evolve Space Solutions, the European Space Agency, Universidade Nova de Lisboa, and Universidade de Coimbra.

The goal of VIRTU is to enable on-demand configuration and deployment of Virtual Machines (VMs) and applications, independently from the vendor, enabling virtual infrastructure management for the IT industry and for the experimental/testing of complex systems [2]. The configuration of VMs at the application level (fine control over the installed applications) provides the means for the fine-grained creation and provisioning of configurable virtualized application stacks.

One of the challenges addressed by the VIRTU project is to enable the configuration of such application stacks in large scale virtualized environments. A process that must be scalable, automatic and executed with no administrator intervention, goals that antagonize with the flexibility required by user customized installations.

Applications are usually parameterizable through configuration files, whose basic concepts crosscut most applications. The absence of an adopted standard representation makes the systemic configuration of computer systems a hard and time consuming task. Automatic configuration can be achieved by processing the system and application configuration file(s) and applying a set of regular expression based search-and-replace operations. However, this approach has several limitations: i) it only applies to text-based configuration files; ii) it limits the scope of the changes to be applied to the scope of the search string, and; iii) it assumes the configuration file keywords and syntax will not change in future versions of the application. In order to overcome these limitations, we propose the Smart Application Reconfiguration Framework (SmART), that automatically configures systems (Including Virtual Machines) and application stacks (possibly inside Virtual Appliances), regardless of the application being configured.

The SmART framework implements an application configuration workflow by recognizing the syntax and, to some extent, the semantic of a configuration file, and producing a structured generic (application independent) equivalent intermediate representation. SmART also embeds the syntax of the original configuration file into the generic intermediate representation. Configuration transformation scripts may operate over the intermediate (structured) representation, with or without administrator support, to safely generate a customized configuration. A generic component uses the embedded syntax information in the intermediate representation to

generate a new configuration file, equivalent to the original one but reflecting the applied customizations.

SmART deals with the heterogeneity of configuration file formats by exploring the similarities between configuration files of different applications. For example, text based configuration files include typically parameter definitions, parameter blocks and comments. In this paper we limit our scope to text-based configuration files, not covering, for now, the processing of binary configuration files.

The remainder of the document is structured as follows: Section 2 analyses the format and structure of the configuration files of widely used applications; Section 3 presents the SmART architecture and concrete implementation; Section 4 shows how we chose to evaluate the tool and reflects on the obtained results; Section 5 discusses the integration of SmART in the VIRTU project; Section 6 covers the related work; and, finally, Section 7 states our final conclusions on the carried work.

2 An Analysis of Application Configuration Files

This section focuses on identifying patterns in the structure and contents of application text-based configuration files, in order to attain a uniform representation. We performed a comprehensive analysis of the configuration files of well known and widely used applications, such as Apache, Eclipse, MySQL, PostgresSQL, GNUstep, and Mantis. The study was, for now, confined to open-source applications. As anticipated, the concrete syntax of configuration files tends to differ, but they resort to a limited number of concepts. In fact, only four distinct concepts were identified in all of the inspected files:

- *Parameter assignment* – set the value of an application configuration parameter;
- *Block* – group configuration settings;
- *Comment* – explain the purpose of one or more lines of the file;
- *Directive* – denote commands or other directives, such as the inclusion of a file.

Our analysis also focused on the actual representation used by the applications to express these concepts. Although no standard exists, we observed that some formats, such as INI [1] and XML [10], have emerged as community standards. Consequently, we were able to classify all studied applications under the following three categories:

INI-based (Listing 1): The syntax follows the INI format or similar. Assignments conform to the syntax "*parameter separator value*", where *value* can be either a single value (line 4), a list of values, or even empty (lines 2 and 3). Blocks are explicitly initialized (line 1) but are implicitly terminated by the beginning of the next block, which discards block-nesting support. For instance, the block that begins at line 6 implicitly terminates the previous block that began at line 1. Comments are denoted by a special character (line 9).

XML-based (Listing 2): This category addresses applications that store their configurations as XML variants, encompassing syntaxes a little more permissive than pure XML. This category has a broader scope than the INI-based, since explicitly initialized and terminated blocks (lines 2 to 4) can nest other blocks.

Block-based (Listing 3): This category addresses a wider scope of formats on which blocks are delimited by symmetric symbols, such as { } or ( ). The depicted example contains three of such blocks. The first one is anonymous and envelops the entirety of the file; the second, NSGlobalDomain, ranges from line 2 to 3, and; and the third, sogod, ranges from line 4 to 11.

Listing 1 MySQL configuration file snippet - INI-based example

```
1  [mysqldump]
2  quick
3  quote-names
4  max_allowed_packet      = 16M
5
6  [isamchk]
7  key_buffer              = 16M
8
9  # The MySQL database server configuration file.
10 !includedir /etc/mysql/conf.d/
```

Listing 2 Eclipse configuration file snippet - XML-based example

```
1 <workbenchAdvisor/>
2 <fastViewData fastViewLocation="1024">
3 <orientation view="org.eclipse.ui.views.ContentOutline" position="512"/>
4 </fastViewData>
```

Listing 3 GNUstep configuration file snippet - Block-based example

```
1  {
2    NSGlobalDomain = {
3    };
4    sogod = {
5        NGUseUTF8AsURLEncoding = YES;
6        SOGoACLsSendEMailNotifications = NO;
7        SOGoAppointmentSendEMailNotifications = NO;
8        SOGoAuthenticationMethod = LDAP;
9        SOGoDefaultLanguage = English;
10       SOGoFoldersSendEMailNotifications = NO;
11   };
12 }
```

The conclusions drawn from our analysis sustain our premises: the basic concepts of application configuration are crosscutting, and thus can be reduced to a structured generic representation, detached from the application specifics. This generic representation can then be modified systemically and once altered be converted back to the original syntax, reflecting the applied modifications.

3 An Application Reconfiguration Framework

This Section presents SmART, a framework that builds on the idea of reducing configuration files to a generic (application independent) intermediate representation, in order to provide scalable means to systematically apply configuration transformations, with or without administrator support, for IT infrastructure administration and maintenance. We will address the framework's architecture, execution flow, extensibility concerns, concrete implementation, and produced output.

As depicted in Figure 1, the reconfiguration process is divided in five steps: i) Identification (and extraction) of the configuration files to be customized; ii) Translation of the configuration file from its original representation to a generic one (stage 1); iii) Customization of the (generic) configuration file to reflect the desired configuration (stage 2); iv) Translation of the generic configuration file back to its original representation (stage 3); and v) Incorporation of the new configuration file into the application/system.

The stages 1 to 3 are independent, requiring only the use of the same intermediate representation. Such design enhances flexibility, for instance: file modification can be performed manually (e.g., a graphic tool that displays all the editable settings) or automatically (e.g., a script); several executions of stage 2 can be performed over a single output of stage 1; and stage 3 can convert a file back to its original representation with no extra information besides the one included in the input file.

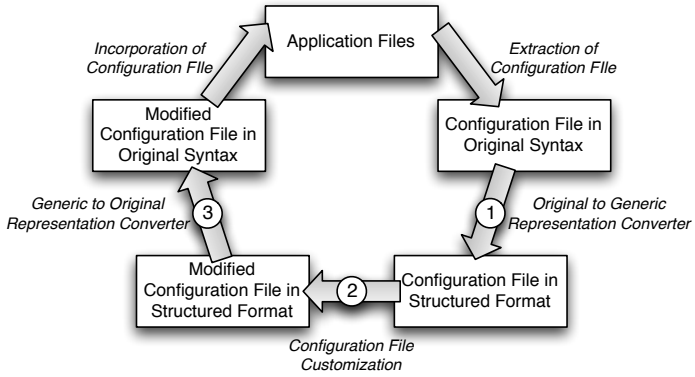


Fig. 1 Configuration file transformation workflow in SmART

The emphasis of this paper is on the systemization of the whole configuration process, namely on stages 1 and 3, covered in Sections 3.1 and 3.2, respectively.

3.1 Original to Generic Representation (O2G)

The purpose of the O2G component is to convert configuration files from their original syntax into the structured generic intermediate representation. The converter must be equipped with a set of parsers capable of recognizing as much configuration files as possible. A given file is translated into an internal uniform representation that is dumped to a file, according to a concrete generic syntax. This syntax is implementation specific and no restrictions are imposed at the architectural level.

Although our analysis revealed that three categories are sufficient to classify the totality of our case studies, it is clear that other formats are or will be used. Thus, for the sake of extensibility, new parsers can be added to the O2G, either directly, as long as framework compliance is preserved (see Section 3.1.3), or by submitting a grammar that the O2G will use to produce the parser itself. Section 3.1.3 also addresses the concern of extending the concepts recognizable in a configuration file.

3.1.1 Architecture

Regarding its internals, the O2G comprises six components organized in a three-tier software architecture as illustrated in Figure 2.

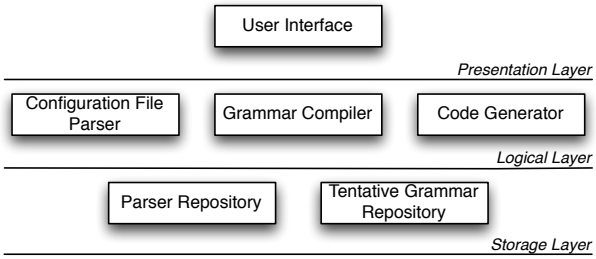


Fig. 2 Original to generic representation converter components

The Parser Repository is the database that stores the parsers currently available to the framework. It provides the means to add, remove and update parsers. The Tentative Grammar Repository stores all the grammars defined by the user on the process of producing a new parser. The repository can be logically viewed as a tree, providing a simple way of iterating previous attempts. The actual parser generation is performed by the Grammar Compiler that resorts to an external parser generator (e.g., JavaCC [5]) to compile the grammar and, consequently, produce the new parser. The return value indicates whether the compilation was successful or not. The Configuration File Parser produces the abstract syntax tree (AST) of a given configuration file. The operation can be performed by all the available parsers or by a given single one. Each parsing attempt produces an output that includes the AST

and the statistics regarding the percentage of file successfully parsed. Naturally, the *all parsers* option produces a list of such results. The AST itself provides an internal uniform representation of a recognizable file. It is composed of nodes denoting the concepts identified in Section 2 (parameters, blocks, comments, and directives) or denoting new concepts introduced by the user. Its output to a file, according a chosen concrete generic syntax, is performed by a specialized implementation of the Code Generator. Finally, the User Interface poses as an intermediate between the lower layers and the user, exposing all the framework's functionalities.

3.1.2 Execution Flow

Figure 3 illustrates interactions between the O2G component modules. Each file received by the O2G is passed to the Configuration File Parser that iterates the Parser Repository to check if, at least, one of the available parsers in the repository is able to perform a successful recognition. If such is the case, the result of the operation is made available to the user, in order to be validated. A positive evaluation will cause the AST to be translated into generic representation via Code Generator, whilst a negative will force the Configuration File Parser to continue its iteration of the repository, until no more parsers are available. When none of the parsers is capable of completely recognizing a given file, the user will have to supply a new one to the framework. This can be achieved by directly importing an already existing parser, or by submitting a grammar to the O2G so that the parser can be internally generated by the framework. To ease the burden of this latter task, statistics are collected for each parsing attempt and the parser that performed better is made available to be used as a working base.

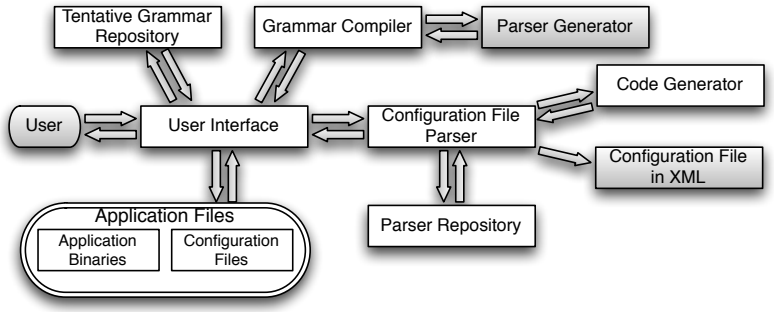


Fig. 3 Original to generic representation component interactions

The Grammar Compiler is the component responsible for producing a parser from a submitted grammar. The parser will be added to the Parser Repository if it fully recognizes the target file and its output is validated by the user.

The whole parser creation process is assisted by the Tentative Grammar Repository that keeps every submitted grammar, allowing for the rollback of modifications

by iterating the previous attempts. As soon as the parser is added to the Parser Repository the Tentative Grammar Repository is cleared.

3.1.3 Extensibility

The import of existing parsers is disciplined by an interface that specifies the framework's compliance requirements, i.e., the Grammar Compiler/parser interaction protocol. This includes the format of the parsing output, which must recognizable to SmART, i.e., comply with the existing AST node types (a direct mapping from the concepts the framework is able to recognize in a configuration file).

The set of the recognizable concepts can also be extended. Once again the compliance requirements are specified by an interface that determines which data extracted from the source file is to be passed to the generic representation.

3.1.4 Implementation

A Java prototype of the SmART framework has been implemented and evaluated. The implementation required technology decisions concerning the concrete syntax for the generic representation and the external parser generator. The choice fell, respectively, on XML [10] for its widespread use and support in mainstream programming languages, and JavaCC [5] for its functionalities and ease of use.

A major design and implementation challenge was how to communicate the details of original file's syntax to the Generic to Original Converter (stage 3), providing it with the information required to later generate a valid customized configuration file from the generic representation. A clean and flexible solution is to code this information in the XML file, along with the generic representation. For that purpose we defined two dedicated XML elements: Metadata and FStr. The first is present once, at the beginning of the file, and contains immutable information about each concept that the file exhibits. The second is a format string present at each concept instance, and specifies how such instance must be written in its original syntax. The string is composed by a list of placeholders that indicate where the actual data to be written is stored. The placeholder possibilities are:

- %a.*x* Print the value of attribute *x* from the current XML element;
- %e Print the value delimited by the next inner XML element;
- %m.*x* Print the value delimited by *x* within the metadata XML element storing information about the concept being processed;
- %c Recursively print the next inner XML element, using its format string;
- %s Print a blank space;
- %n Issue a new line.

We exemplify the XML produced output with the translation of part of the MySQL configuration file snippet of Listing 1. The generated metadata is presented in Listing 4, while Listings 5 and 6 contain, respectively, the translation of the block

ranging from lines 1 to 4, and of the comment in line 10. Space restrictions do not permit the exposure of complete real-life examples, but these can be found in [6].

Listing 4 Metadata

```
1  <Metadata>
2    <Comment>
3      <start>#</start>
4    </Comment>
5    <Parameter>
6      <equal>=</equal>
7    </Parameter>
8    <Block>
9      <start>[</start>
10     <end>]</end>
11   </Block>
12 </Metadata>
```

Listing 5 XML generic representation of a block

```
1  <Block name="mysqldump">
2    <FStr>%m.start%a.name%m.end%n%c%c%c%c</FStr>
3    <Parameter>
4      <FStr>%e%n</FStr>
5      <Key>quick</Key>
6    </Parameter>
7    <Parameter>
8      <FStr>%e%n</FStr>
9      <Key>quote-names</Key>
10   </Parameter>
11   <Parameter>
12     <FStr>%e%m.equal%e%n</FStr>
13     <Key>max_allowed_packet</Key>
14     <Value>16M</Value>
15   </Parameter>
16 </Block>
```

Listing 6 XML generic representation of a comment

```
1  <Comment>
2    <FStr>%m.start%e%n</FStr>
3    <Text>The MySQL database server configuration file.</Text>
4  </Comment>
```

3.2 Generic to Original Representation (G2O)

G2O performs the operation complementary to O2G, by converting configuration files from their generic representation back into their original syntax. It is composed of a single component which is able to reconstruct the configuration file by processing the metadata available in the generic representation. The module traverses the XML representation, outputting each concept according to the specified format string and metadata information.

Observe the XML generic representation depicted in Listing 5. The output format string for the block (line 2) is "%m.start%a.name%m.end%n%c%c%c%c" with the following meanings:

- %m.start – inspect the metadata element holding information about the concept current being processed (a Block) to retrieve the data delimited by start, i.e., [;
- %a.name – retrieve the value of the name attribute of the current concept, i.e., mysqldump;
- %m.end – retrieve] from the metadata;
- %n – issue a new line;
- The three %c – process the next three inner blocks (all of type Parameter in this example) according to their own format strings.

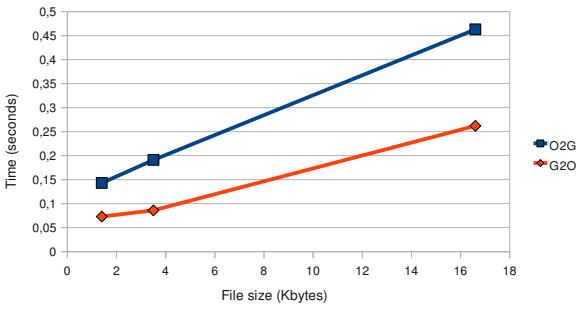
The output will be equivalent to the original contents displayed in Listing 1.

4 Evaluation

The framework was evaluated from three different angles: functionality, operationally and performance. The functional evaluation aimed at verifying if the requirements towards which the framework was designed were all met. For instance, a requirement was that *the framework should be extensible to accommodate configuration files with new/different syntaxes*. This was accomplished by featuring a process on which a user is able to build parsers by submitting grammars, or simply by adding a new parser to the ParserRepository, as long as it complies with the framework's parsing invocation protocol. The comprehensive requirement list can be found in [6].

The objective of the operational evaluation was to determine if the framework can correctly transform configuration files into the XML generic representation and back to their original syntax. Several tests were performed upon real configuration files [6] of the three categories presented in Section 2. The initial and final files, with no customization in the intermediate representation, were compared using the UNIX *diff* utility. Overlooking ignorable characters, such as extra black spaces in the considered formats, all compared files were identical.

The performance evaluation focused on the time required by the O2G and G2O converters to apply their transformations. A first test aimed at assessing if the time required by the transformations is proportional to the size of the source file. The graphic displayed on Figure 4 refers to the processing of INI-based PostgreSQL configuration files (available at `http://asc.di.fct.unl.pt/smart`).



	O2G	G2O	Total
File 1.4KB	0.143s	0.073s	0.216s
File 3.5KB	0.191s	0.086s	0.277s
File 16.6KB	0.463s	0.262s	0.725s

Fig. 4 Performance evaluation

We can observe, the processing time in both converters grows linearly with the size of the file to be processed, but in both cases is suitable for an interactive tool. The largest evaluated file, of 16.6 KB, spent only 0.46 seconds on stage 1 and 0.26 seconds on stage 3.

A second test focused on another aspect worthy of monitoring. The time required to generate a parser from a given grammar. Table 1 presents the time spent by JavaCC to generate the parsers for the three format categories of Section 2. The results indicate that parser compilation time is also sustainable. Nevertheless, to

have a more accurate assessment we compiled a parser for Java, a language likely to be more complex than any configuration file format. The elapsed time was 0.55 seconds, which sustains the previous conclusions.

INI-based	Block-based	XML-based
0.27s	0.28s	0.30s

Table 1 Elapsed times for parser compilation in JavaCC

Both tests were performed on a system comprising an Intel Pentium Dual T3200 processor with 2GB DDR2 main memory, and running the Linux 2.6.31 kernel.

5 VIRTU Integration

This section briefly discusses the integration of SmART in VIRTU [3], a platform conceived to enable on-demand configuration and deployment of VMs and application stacks independently of the vendor.

VMs in VIRTU are constructed by assembling *building blocks* (operating system, and applications) whose configuration is specified in special purpose files, named *publication files*. Configuration is decoupled from assembling, allowing many-to-many relationships. Thus, a VM is only configured when deployed. The publication files specifying the configurations of the assembled building blocks are retrieved from a pre-determined database and handled by a script, to perform the desired configurations before the system is on-line.

SmART integrates with VIRTU at two levels. The O2G converter is used by the administrator to transpose one or more configuration files into a building block publication file, in order to define the block's default configuration and user editable parameters. The G2O is used in the VM's on-boot configuration process. It converts the building block's publication file back into its original format, so that the configuration script may carry its work. Reconfigurations can also be performed by having a daemon (on every running VM) listening for reconfiguration requests. The process is equivalent to the on-boot configuration process, with the exception that the target application may have to be restarted.

6 Related work

To the best of our knowledge, our approach is the first to exploit the similarities among configuration files to allow for automatic, vendor-independent and on-the-fly application reconfiguration. Similar existing projects, such as AutoBash [8] or Chronus [11], take on automatic application configuration as a way to assist the removal of configuration bugs. AutoBash employs the causality support within the Linux kernel to track and understand the actions performed by a user on an appli-

cation and then recurs to a speculative execution to rollback a process, if it moved from a correct to an incorrect state. Chronus, on the other hand, uses a virtual machine monitor to implement rollback, at the expense of an entire computer system. It also focuses a more limited problem: finding the exact moment when an application ceased to work properly.

Two other projects, better related to this work, are Thin Crust [9] and SmartFrog [4]. Both projects aim at automatic application configuration, but take an approach different from ours. Thin Crust is an open-source set of tools and metadata for the creation of VAs. It features three key components: Appliance Operating System (AOS), Appliance Creation Tool (ACT) and Appliance Configuration Engine (ACE). The AOS is a minimal OS built from a Fedora Kickstart file, which can be cut down to just the required packages to run an appliance. SmartFrog is a framework for the creation of configuration-based systems. Its objective is to make the design, deployment and management of distributed component-based systems simpler and more robust. It defines a language to describe component configurations and a runtime environment to activate and manage those components.

7 Conclusions and Future Work

The work described makes evidence that systemic configuration of applications can be safely achieved by abstracting configuration files from their format specificities. The SmART framework enables this idea by featuring two complementary modules (O2G and G2O), that perform transformations between the application dependent syntax and a generic representation and back, regardless of the original application.

A proof-of-concept prototype has been implemented. By default it supports the three format categories that, according to the analysis in Section 2, cover the majority of the existing applications with text-based configurations files. Nonetheless, a major effort was put on extensibility. The framework allows for the addition of new configuration file parsers, and of the concepts that can be recognized on a file.

The carried evaluation certifies that all of the tested use case files can be transformed into our generic representation and back. Moreover, both these operations (as well as grammar compilation) are performed in a time that is adequate for an interactive application, i.e., less than one second. The prototype is being integrated into the VIRTU product line and is being further extended by Evolve.

We can therefore conclude that, once having a parser able to recognize the source configuration file, SmART contributes to accelerate the configuration process, since it does not require the knowledge of the source file format in order to apply the desired modifications. Moreover, the use of a generic representation allows for the systemization and automation of the whole process.

Regarding future work, naturally that there is room for improvement at different levels. For instance, to broad the scope to include binary files, which may even require the addition of new recognizable concepts. However, in our opinion, the main research challenge is on the use of grammar inference [7] to create new parsers.

By inferring grammars, instead of delegating their definition on the user, the creation of new parsers can be perform almost entirely without the user's intervention, enhancing usability and generality.

Acknowledgements This work was partially funded by ADI in the framework of the project VIRTU (contract ADI/3500/VIRTU) and by FCT-MCTES.

References

1. Cloanto: Cloanto implementation of INI file format. `http://www.cloanto.com/specs/ini/` (2009)
2. Duro, N., Santos, R., Lourenço, J., Paulino, H., Martins, J.A.: Open virtualization framework for testing ground systems. In: PADTAD 2010: Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. Trento, Italy (2010)
3. Evolve Space Solutions: VIRTU Tool. `http://virtu.evolve.pt/` (2009)
4. Goldsack P. et al: The SmartFrog configuration management framework. SIGOPS Oper. Syst. Rev. **43**(1), 16–25 (2009)
5. Kodaganallur, V.: Incorporating language processing into Java applications: A JavaCC tutorial. IEEE Software **21**, 70–77 (2004)
6. Martins, J.: SmART: An application reconfiguration framework. Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (2009)
7. Parekh, R., Honavar, V.: Learning DFA from simple examples. Machine Learning **44**(1/2), 9–35 (2001)
8. Su, Y.Y., Attariyan, M., Flinn, J.: AutoBash: improving configuration management with operating system causality analysis. SIGOPS Oper. Syst. Rev. **41**(6), 237–250 (2007)
9. Thin Crust: Thin crust main page. `http://www.thincrust.net/`
10. W3C: Extensible Markup Language (XML). `http://www.w3.org/XML/`
11. Whitaker, A., Cox, R.S., Gribble, S.D.: Configuration debugging as search: finding the needle in the haystack. In: OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, pp. 77–90. Berkeley, CA, USA (2004)