

review the technology which is currently available to solve them, and finally motivate our dissatisfaction with the development process which is required by the current technology.

2.1 Autonomic problems

The deployment of autonomic technology in an organization's IT infrastructure can be an important factor in ensuring reduced management costs, increased reliability and availability of the infrastructure, and improved response time in the case of failures or external attacks to the IT resources. Yet, each organization's needs and priorities are different, and there is no such thing as a "one size fits all" autonomic solution. Hence, before a comprehensive and reliable autonomic management solution can be set up, extensive consulting with the various decisional and technical levels in the organization must be conducted, and possibly several prototypes must be produced, deployed, and tested on the field to collect enough feedback to drive final development, in what is typically a spiral development model, based on extensive prototyping.





To keep the presentation simple, we will focus here on a specific sub-topic of some relevance in autonomic systems: problem determination and log analysis. Problem determination focuses on identifying the causes of any deviation from the expected behavior of a system. Typically, this is performed by directly observing the system behavior, comparing expected performance to observed one, and once a deviation is noticed, hypothesizing the cause, based on knowledge of the system's structure and internals. A more effective approach involves observing not only the external behavior of the system, but also that part of its internal behavior which is exposed by its various components through so-called *log files*¹; thanks to the additional information made available through log files, both symptoms and diagnosis can be more precise, and ultimately any action taken to remedy the deviation can be more timely and focused on the real problem. As the simplest of examples, consider the case where a complex system simply stops performing completely: based purely on external observation, one could try to restore the system to functioning by shutting it all down and then restarting it, whereas with more precise observation coming from log files, a system manager could notice that only a specific sub-component or service is down, and can restart only the relevant subsystem, possibly reducing downtime considerably.

The analysis of log files, from various sources, in order to discover any problem, or simply to measure performances, is called *log analysis*. Log analysis can be described as the search for relevant *patterns* in complex log data. Despite being conceptually simple, in practice log analysis often presents unexpected problems. First, and most relevant, which patterns is worth or necessary searching for (and are thus relevant) is typically far from obvious. As happens for most other *information retrieval* and *extraction* tasks, there is a definite trade-off to


¹Which, given the current technology, may not be files at all, but made available to inspections through various kind of programmatic interfaces, or broadcast on a common event bus.

be found between *completeness* (we want all the interesting patterns to be captured) and *relevance* (we want the amount of data produced by analysis to be manageable). Second, syntactic and semantic differences in log data can make analysis a challenge. For example, different subsystems may refer to the same concept in different ways in their log data, or refer in the same way to slightly different concepts. A number of technological issues, such as format of the log file and clock skew between different systems and machines, also come in the way.


2.2 Current technology

Several years of industrial and academic research have produced solid, working solutions* for the two problems mentioned above. The second one, that of log heterogeneity, has been addressed by IBM*  the CBEM (Common Base Event Model) standard model for log events, and by providing *log parsing* technology for converting logs in a variety of format to CBEM. A *log parser* in the Autonomic Toolkit  small software module (written in Java)  which reads log records in a custom format, and emits corresponding records in CBEM format. The translation may both add information (e.g., identity of the reporting modules, which the original application, writing on its own log file, may have considered redundant) and lose some (e.g., details for which no corresponding CBEM property has been defined). In such cases, it is customary to put any extra information in a textual, free-form "message" field in the CBEM record. 

* mention Autonomic Toolkit
* Check authors and add reference

The first problem, that of identifying relevant pattern, has been addressed in a similar way — by assuming that a software module, again implemented in Java, will sift through the various log flows and return all the relevant patterns, or *correlations* in autonomic parlance, found therein. However, while the conversion of log format to CBEM is a one-time job, which can be assumed will be performed once and for all by a competent developer, finding correlation is a continuous activity. The notion of what is a *relevant* pattern of events changes depending on protean business needs, varying pattern of attack from malicious third parties, different level of details requested, shifting priorities, required quality-of-service levels which change depending on market factors, and so on. The assumption that correlations will be found by writing a piece of software anew for each possible pattern of interest is thus quite burdensome to final users, who may even be (and often are) incapable of performing the task themselves. This situation leads to the (unsatisfactory) current development process we will describe in the next section. 

2.3 Current process

In current practice, the development of new *correlators*, as the software modules responsible for identifying patterns are known, is typically conducted jointly by the customer (with representatives both at the managerial and at the technical levels) and by a consultant (see Figure 1)  standard requirements elicitation approaches are used (e.g., interviews, brainstorming, mock-ups) to elicit the goals

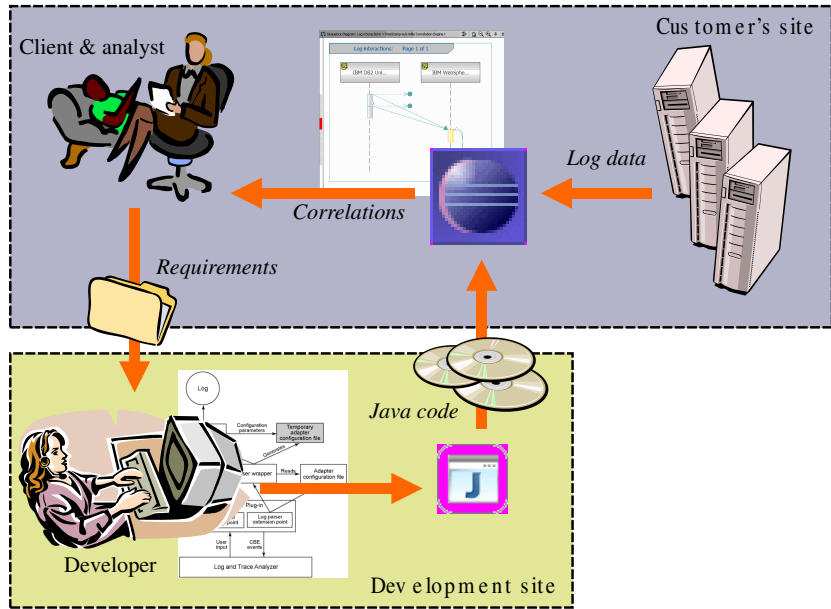


Figure 1: The current process for developing a log correlator.

of the customer, then the consultant, with help from the customer's technical personnel, formulates a set of informal guidelines which are then sent, together with some sample data, to a remote development site for writing the relevant log correlators (and often, any needed log parser). The software comes back from the development site for deployment and testing at the customer's site, where it is verified. Typical cycle time for this activity has been reported to be about a week*, although exact timing may vary depending on the complexity of the correlation rules to be developed, on the priority assigned to the customer, etc.

* Stefano: any source for this?

Most often, the development and refinement of a correlator will take several rounds, as the customer clarifies his own needs, the consultant gains a better understanding of the customer's goals, and the developer embodies precisely all the details in the code. The time from initial inception to final delivery may be thus several weeks, during which the customer's needs will not be satisfied. Moreover, whenever these needs change (for any of a number of reasons, as discussed above), or new needs arises, the process will have to be repeated. If, for example, the customer deploys a new subsystem in its IT infrastructure (say, load-balancing hardware), the set of log flows to be analyzed changes, and new analysis opportunities arise, opening the scope for updates to previously developed correlators, and for developing new ones.

This substantial cycle time we regard as unsatisfactory, especially in view of the general purpose of autonomic technology in general: reacting automatically, quickly and effectively to difficult circumstances, like a breakage or an external

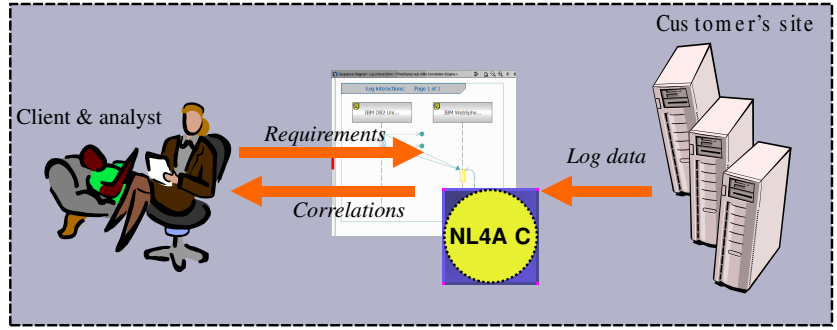


Figure 2: The correlator development process with the NL4AC approach.

attack to the IT infrastructure. It is precisely under such circumstances that timeliness is more important.


3 The NL4AC approach

The problems with the current process, which we described above, can be ascribed mainly to a classic phenomenon of *communication impedance*: the language used by the customer to express his goals, concerns and problems (typically, English and possibly some diagram) and that required as input by the autonomic system (Java source code) are too far apart. This language mismatch opens the scope for ambiguities, misinterpretations, delays which are one of the root causes for the need of several iterations of the process. Moreover, the task of writing log parsers and correlators directly in Java, while not terribly difficult from a technical point of view, is often out of reach for many decision-makers in the customer company, which are the ones whose goals the autonomic system should try to satisfy. Hence, a number of different subjects have to join in the process, which again slows down “rapid” prototyping approaches.



Questo paragrafo é veramente importante.

We address this problem by changing the languages that are used. In particular, we advocate that using *controlled natural language* to capture customers’ goals *and* for generating the relevant software modules, the development process cycle time can be radically reduced, making truly interactive development of autonomic solutions feasible, while at the same time reducing the risk of error injection due to communication impedance. The process we envision is thus the one depicted in Figure 2.

As a proof of concept, we have in fact developed such a solution for log correlation, which we will describe in some detail in this section. The natural language processing technology we use  has been deployed in a set of plug-ins for the Autonomic Computing Toolkit, called the NL4AC Plug-ins for Eclipse. Field validation of the proposed technique will be discussed in the next section, after presentation of the technology.

3.1 Features description

The NL4AC plug-ins for Eclipse provide an environment for writing *correlators* for the *Eclipse Test and Performance Tools Platform* or *TPTP* (which include the *Log and Trace Analyzer* framework). In particular, NL4AC allows correlators to be written in (controlled) natural language, rather than as Java code. The NL4AC plug-ins implement a framework and architecture which is fairly general and vastly extensible, as will be pointed out later in Section 3.5; many aspects of their operations are controlled via configuration files or sub-plug-ins². The configuration provided in the prototype implementation allows end-users to describe the desired correlation in terms of relationships and predicates on the values of attributes of the CBEM model.

The NL4AC plug-ins offer the following services:

- creation of a NL correlation project;
- parsing and analysis of NL³ text;
- feedback on the results of the parsing process;
- synthesis of Java code for a correlator based on the NL input;
- “live” testing of the correlator on real data;
- export of a TPTP correlation plug-in with the generated code.

These services are provided in the context of the standard document editing model of Eclipse, and the operations of the NL4AC plug-ins mimic as completely as possible the usual edit/compile/get feedback/test cycle used for code development.

The plug-ins also provide a new document type (with extension `.corr`), a wizard to create such documents, a corresponding editor with completion and syntax coloring, and problem markers to report on parsing and analysis problems.

It is worth noticing here that although the NL4AC system itself is embedded in the Eclipse integrated development environment, it is not aimed directly at developers only; on the contrary, we assume that a consultant at the customer site, sitting next to his customer, will be the most typical user of the technology.



3.2 Architecture description

In this section we describe the high-level architecture of the NL4AC system in terms of its logical components, allocation of components to plug-ins, and interaction between these components during operations.

²Notice that these sub-plug-ins are not implemented as Eclipse extension points; rather, the more specialized `CIRCE` [2] extensibility architecture is used.

³The current configuration supports English as native language.

3.2.1 Logical components

The NL4AC system consists of four logical components, namely

- the **CICO** domain-based parser, which is used to parse English text;
- the **CIRCE** modular expert system, which allows the system to reason on and augment the results of the parsing;
- the **NL4AC/Eclipse** subsystem, which implements the interface between Eclipse and the other components, and drives the overall computation, and
- the **Delegating Log Correlator**, which by implementing lazy dynamic class loading and a delegation-based correlation engine allows the end user to immediately test the correlation that is being developed without having to deploy a correlation plug-in for each test.

Of these, the CICO and CIRCE components were developed prior to and independently from the NL4AC project; full documentation about them can be found in [3, 4, 2]. While the basic design of both components has remained the same, they have been re-implemented for the NL4AC system. In particular, CICO has been packaged as a OS-specific library (Linux and Windows versions are provided) accessed through JNI*, while CIRCE has been rewritten in Java.

* Ref to JNI

The NL4AC/Eclipse and Delegating Log Correlator components are documented in the following.

3.2.2 NL4AC/Eclipse interface

The entire interface between Eclipse and the NL4AC system is implemented in the NL4AC/Eclipse component. The component introduces a new file type, the `.corr` file, which is an XML document containing, in addition to identification and other administrative data, three textual elements: the NL description of the desired correlation (*text*), an optional set of additional designations (*glossary*) which list names and synonyms for special entities in the domain (e.g., alternate names for a given host or application), and an optional set of *definitions* which allow expert users to define additional fragments of the accepted language (e.g., jargon or concise abbreviations). In basic use, only the text part is needed, that is, users need not see or modify the glossary and the definitions.

The component also provides the following extensions:

- A “new file” wizard to create new `.corr` files from a template;
- A file editor which allows users to edit the contents of `.corr` files; the editor offers three tabs for editing, respectively, the text, the glossary and the definitions of a document.

- A builder⁴ which uses the CIRCE engine to parse and analyze the text, and synthesize corresponding Java code fragments, which are then injected in a predefined template to obtain the full code for a correlator.
- A class of problem markers, to report about parsing and validation problems found during the build; these markers are prominently displayed in the user interface by the Eclipse infrastructure.
- A project nature, which connects a project to the builder (lacking this connection, `.corr` files are treated as regular text files).
- An action contributor, which adds various UI elements (including pop-up menus entries and toolbar buttons) to perform various operation such as toggling the NL4AC nature of a project, dumping debug information, etc.

3.2.3 Delegating Log Correlator

The Delegating Log Correlator implements a basic log correlator which simply delegates the actual correlation to a different class, namely the correlator class automatically generated from the natural language text. The delegate class' bytecode is loaded dynamically at each instantiation through a custom class loader; the latter is notified by the NL4AC/Eclipse interface builder when a new correlator class is generated.

Loading of the newly generated class is postponed until actual use by adopting a lazy notification/loading policy. In particular, the NL4AC builder notifies the Delegating Log Correlator of the availability of a new version of a synthesized correlator class as soon as generation of the Java code is completed. The class is actually loaded later on, when the TPTP instantiates the Delegating Log Correlator as part of the creation of a new correlation.

It is worthwhile to remark that this scheme implies that at any given moment, only one correlator can be tested through the Delegating Log Correlator. This is not a problem in typical usage, and furthermore it is necessary due to the lack of any form of explicit parameterization of correlations in the TPTP framework and UI.

3.2.4 Packaging

The four components of NL4AC are packaged in two distinct plug-ins*, as shown in Figure 3.

The CICO parser, the CIRCE subsystem and the NL4AC/Eclipse interface are contained in the main plug-in `it.unipi.di.nl4ac`, which provides most of the features. In addition to the executable native and Java code for the various components, this plug-in also includes configuration files used for customizing the system. The latter include resources for configuring the language parsing

* We may want to drop this part

⁴In Eclipse terminology, a builder is a software component which goes over a document to generate any derived element; compilers and pre-processors are typically implemented as builders.

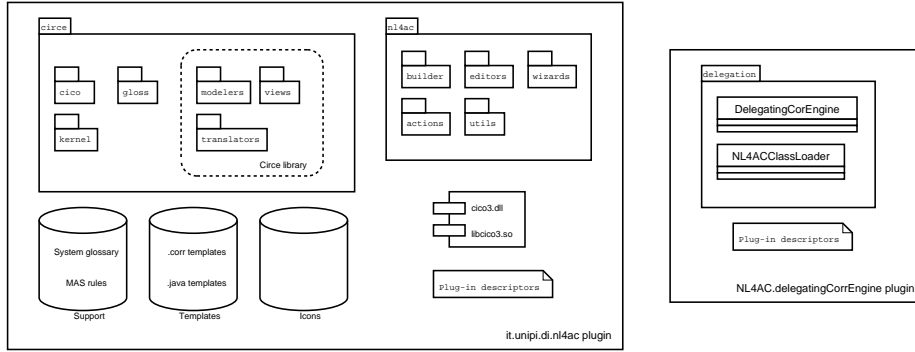


Figure 3: Packaging structure of the NL4AC system.

and analysis steps (in the **Support** folder), templates used to create new NL correlation resources and to generate Java code (in the **Templates** folder), and the *CIRCE library* of modules for the embedded expert system (see [2] or [3] for details on these modules).

The Delegating Log Correlator, together with the associated lazy class loader, are contained in a separate Eclipse plug-in whose plug-in descriptor specifies the needed extensions to the TPTP so that this plug-in is recognized as a correlator by the TPTP.

3.3 Operations

In this section we illustrate how the various components of the NL4AC system interact to realize the main operations offered to the user. Major operations are described through a UML-like sequence diagram⁵ and commented with reference to their user interface.

3.3.1 Creating a new correlation file

Creation of a new `.corr` file (see Figure 4) is invoked through the usual **New** menu entry by selecting the **NL Correlation** creation wizard. The wizard presents the user with a dialog asking for certain parameters (e.g., the name for the new NL correlation), and when all parameters have been validly collected, initiates the creation process.

`.corr` files are created based on a template which can be a standard one, provided with the plug-in, or a custom one, which in turn can be user-specific or site-wide (possibly prepared by an administrator). Once the template is obtained, a file with the given name and contents is created in the workspace, and an instance of the appropriate editor is opened on the new file (see Figure 5).

⁵These diagrams are to be considered informal in nature; we will take some notational liberty in order to simplify the presentation.

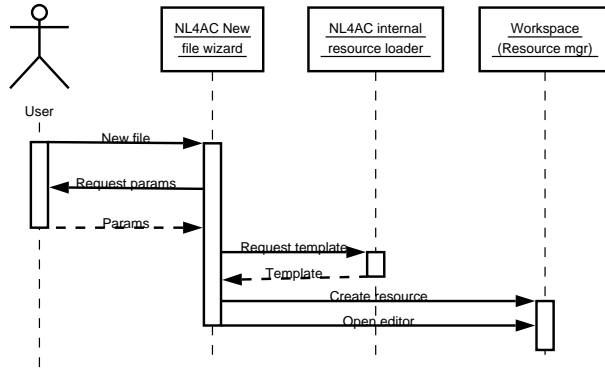


Figure 4: The user invokes the NL4AC New File wizard.

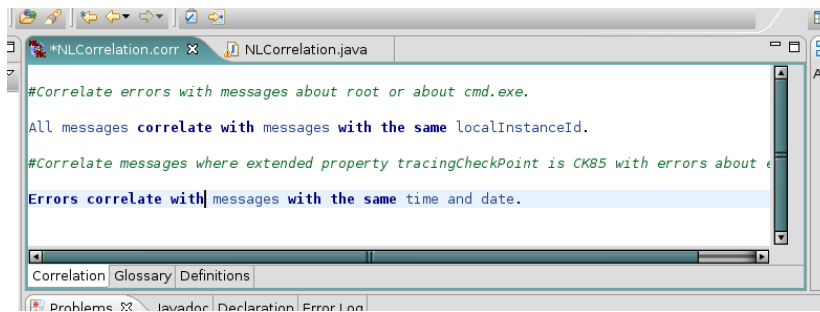


Figure 5: The editor used in the NL4AC system.

The whole creation process is consistent with the way other “New file” wizards work in Eclipse.

3.3.2 Editing a .corr file

Once a correlation document has been created, editing occurs in a standard text editor within the Eclipse IDE (Figure 5). English sentences are written down, specifying which records or events have to be correlated. In the course of the editing, several facilities are provided to simplify the task:

- Syntax coloring is performed by implementing a dynamic lexical analyzer which accesses the system and document glossary (with appropriate caching to reduce overhead) and compares typed text with terms from the glossaries. In addition, quoted text, comments and directives (see [4] for details) are highlighted with different colors.
- Completion is obtained similarly by presenting the user with all terms from the glossaries starting with the current prefix (Figure 6). The CICO parser supports features like *speculative tagging* which would allow more



Figure 6: Interactive completion assists the user in writing correlations.

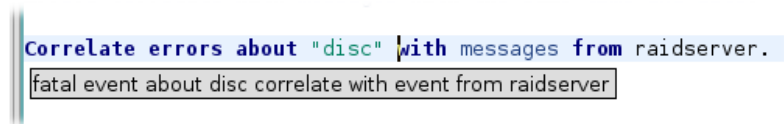


Figure 7: Real-time paraphrase shows how the system has interpreted a sentence.

sophisticated forms of completion to be implemented, but these features are not exposed to the user in the current release.

- Text hovering is used to present the user with a paraphrase of the current text. When the mouse pointer hovers over a sentence, the CICO parser is invoked on the corresponding text, and a linearized form of the returned parse tree is shown to the user (Figure 7). In practice, this provides a way to get feedback, as a paraphrase, on how the system interprets a given sentence. It can be noted that the paraphrase is based on the current text in the editor, not on the version stored in the workspace, so the feedback can be obtained in real-time without any need to save the document.

3.3.3 Saving a .corr file

When the user saves a modified `.corr` file, a complex chain of events is fired (see Figure 8). As soon as the editor saves the modified resource into the workspace, the latter notifies all registered builders for the project which, if the project has the NL4AC Nature, include the NL4AC Builder. The builder reacts by retrieving the new NL text for the correlation, passing it to the CIRCE kernel, and requesting — again to the CIRCE kernel — that the corresponding Java code be recomputed. As soon as the kernel has recomputed the Java code corresponding to the NL text in the saved file, the builder creates a new Java class with that code, and notifies the (lazy) Delegating Class Loader that a new version of the class is available. If CIRCE has reported any parsing or validation

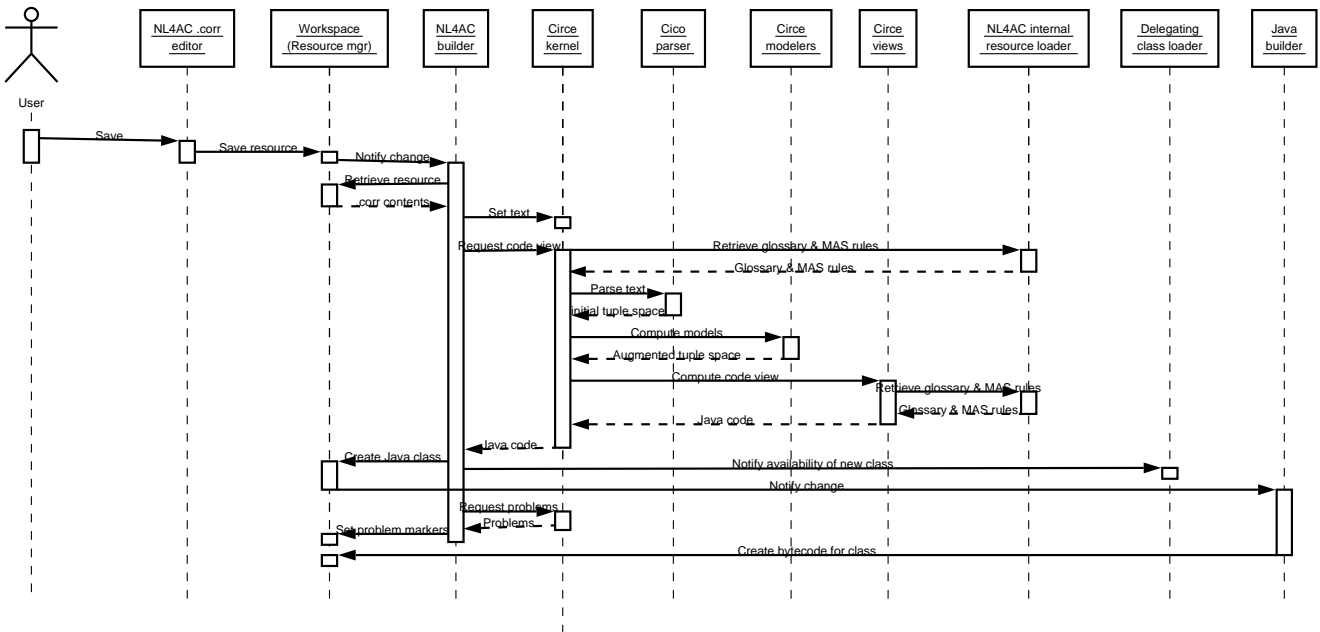


Figure 8: The user save changes to a .corr file.

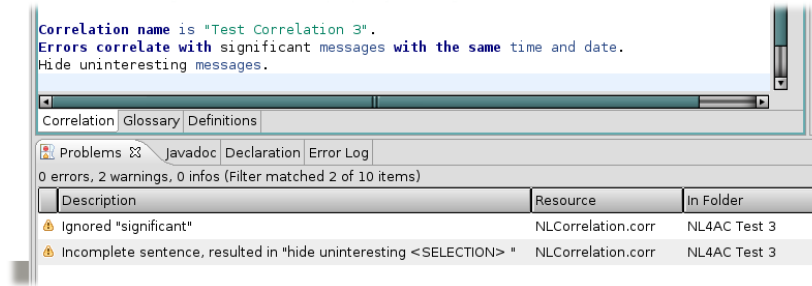




Figure 9: A report of problems found in the analysis of a correlation text.

problem with the text, corresponding problem markers are set on the `.corr` resource (Figure 9). 

Moreover, upon creation of the class the Java builder is notified by the resource manager, so that the new class is compiled in the background, and the corresponding bytecode is stored in the workspace (ready to be loaded, possibly, by the Delegating Class Loader at some future time).

To generate the Java code, the CIRCE kernel first retrieves the system glossary and parsing rules from the plug-in's resources; merges them with those provided in or referenced by the `.corr` file, and then invokes the CICO parser to obtain a parse tree (encoded as a tuple space, as customary in the CIRCE architecture) for the given text. This tuple space is then augmented by invoking CIRCE's expert system modules, or *modelers*, which provide the intensional knowledge of the domain needed for code generation; the augmented tuple space is then finally passed to the code generation view (itself a module from CIRCE's library) which injects the generated code into the chosen template retrieved from the plug-in's resources. The net result of this process is the complete source code for a Java class, which is returned to the builder and then used to create the class in the workspace as described above.

3.3.4 Testing a correlation

To test a newly developed correlator (see Figure 10), the user creates a new correlation based on the Delegating Log Correlator. When the TPTP invokes the `correlate()` method of this correlator, the actual correlation is delegated to the newly developed class, which is loaded dynamically through a custom class loader. The class loader loads the designated class (as established at the "Notify availability of new class" step in Figure 8) directly from the workspace, without employing any form of caching, which guarantees that the latest version of the class under development is loaded every time. Loading of all other classes is delegated to the parent class loader (which is typically the Eclipse standard class loader). 

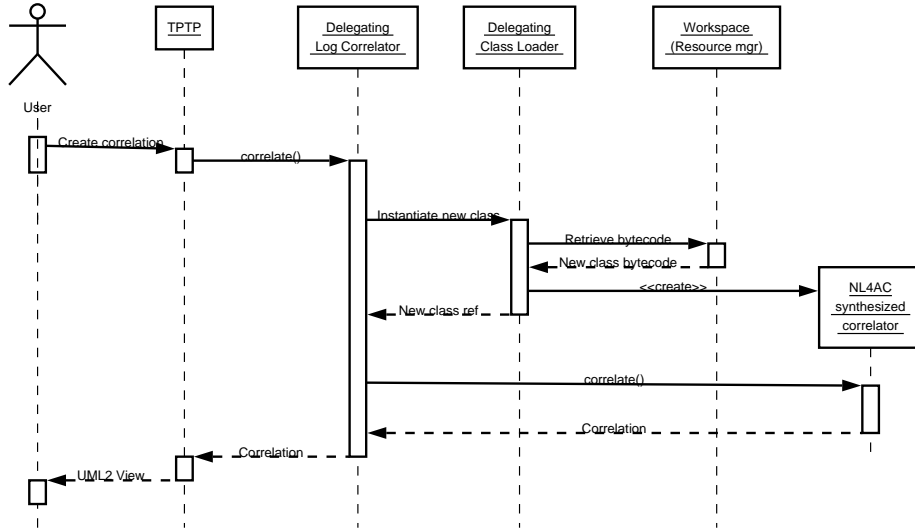


Figure 10: The user tests a newly developed correlator.

3.4 Language

The language recognized by the NL4AC system is tailored to express correlations, i.e. to recognize specific patterns of occurrence of events, as reported in the various log files. Each sentence describes a condition that, if satisfied, will establish a correlation between two log records. Different sentences are considered alternatives; in other words, conditions expressed by different sentences are *or*'ed together in the final correlation.

Each sentence has a structure of the form

selection correlate with selection.

where each *selection* selects a subset of all records, typically based on the values of some field in their CBEM representation. All CBEM fields are supported, and can be referenced by their original name or by a number of common synonyms; further synonyms can be defined in a user glossary.

A number of relational operators are supported, including equality, greater than, greater or equal, less than, less or equal, inequality, substring matches, temporal ordering, etc. Conditions can be composed via logic operators (e.g., *and* and *or*). Abbreviated expressions can be used or defined (via definitions) for common conditions. For example, *warnings* can be used as a short-hand for *events where severityLevel is warning*, and *events from Apache* as a short-hand for *events where applicationId is Apache*. The special form “*attribute of the other event*” is used to bind to the value of the same *attribute* in the event which is being tested for correlation. Fuller details about the language defined by the initial rule set (which, as we will see in Section 3.5, can be extended by administrators and users) are provided in the companion report [1].

It is important to stress that the CICO parser used by the NL4AC is geared towards parsing natural language, not formal languages. Hence, the brief indications given above should not be considered as the definition of *the* recognized language. Rather, CICO works in an information extraction perspective: it tries to extract meaning from a text, and accepts imperfect matches (due, for example, to the presence of additional text, or word order inversion, or even missing parts), hence the accepted language, although controlled in a sense, can be defined only fuzzily. As an end user, looking at the paraphrase that is shown by the editor when the pointer hovers for a second over a line is the best way to check the way the text has been “understood” by the system (see Figure 7).

3.4.1 An example

Let us assume that in a departmental IT infrastructure, the main web server is hosted on a machine called “margot”, the database is hosted on “tera”, and there are a number of other applications and subsystems running on these and other machines. We want to correlate messages referring to the web server, from any applications, with errors reported from the host “margot”, and messages referring to the database (or “db”) with warnings from the host “tera” (we know that “tera” is more fragile, so we want to catch warnings in addition to errors). Moreover, as a general rule we want to cluster together groups of messages coming from the same application in a given time span (say, in the previous 5 minutes) before any error. These business rules can be plainly expressed by writing in the NL4AC editor

```
Correlate errors from margot with messages about "web server".  
Warnings from tera correlate with messages about "database"  
or about "db".  
Errors correlate with events with the same ApplicationID in  
the previous 5 minutes.
```



From this exact text (or from any of a number of alternative forms), the NL4AC can generate the Java code for a correlator satisfying the stated business rules. More importantly, the business rules themselves can be tested interactively, to validate them, and any refinement can be tried out immediately by editing the text.

3.5 Customizability

Autonomic technology has to be tailored to each specific IT environment to be most effective, and log analysis is no exception to this general rule. There is ample scope for customization in the NL4AC system, with a view to providing to end-users a system that has been, or can be, adapted to their specific needs.

Customization in NL4AC can happen at various levels, which we will describe briefly in the following (full details are in [5]).

- The end-user can customize the language accepted by the system by writing *definitions*, which are lax rewriting rules used by the CICO parser in

addition to basic parsing rules. For example, an end-user could write a definition like:

```
CASES WHEN  $x$ /HOST IS DOWN  $\rightarrow$  WARNINGS FROM PINGER ABOUT
 $\$x$ .
```

After such a definition, the user can use the new form (on the left side of the arrow, where x can be substituted by any host name) with the semantics given by the form on the right side. In other words, a sentence like

```
Correlate cases when margot is down with messages about
"web server".
```

generates the same correlator as the sentence

```
Correlate warnings from pinger about margot with messages
about "web server".
```

Any number of definitions can be provided, tailoring the language accepted by the system to the specific needs and habits of each user. The parser will combine the definitions with its own internal rules, applying them as needed to obtain a parse tree for the sentence.

- A site administrator or power user can further customize the system by preparing specialized templates, which can incorporate definitions, designations (i.e., sets of synonyms for local concepts, e.g. host names) or even skeleton sentences. These templates can then be selected by end-users upon creation of the correlation document (as already discussed in Section 3.3.1). Moreover, an administrator can modify the Java code templates which are used to generate the final correlator. Different code templates can provide, for example, varying levels of optimization or user feedback.
- A solution developer can modify the basic set of parsing rules used by the parser, thus changing in arbitrary ways the language recognized by the parser. It should be noted that, while definitions provide new syntax for the same semantics, basic parsing rules can describe language forms whose semantics could not be expressed by pre-existing rules.

A developer can also write new modules for the modular expert system (the CIRCE component of the architecture) used by NL4AC. These modules can be used to encode the semantics for new rules, and how they should be translated into Java code, or to provide additional services like validation of the text, cost estimation for the corresponding correlator, etc.

We believe that only by providing ample opportunities for customization, and at different times and levels of competence, can specialized solutions be applied successfully to different organizational and technical situations.

4 Industrial validation

To write based on feedback from IBM labs.

5 Related work

To write or maybe to drop — I could not find relevant references, different user interfaces for log analysis seems to be a not well-researched topic.

6 Conclusions and future work

Autonomic technology bears big promises, but the issue of how users can best interact with autonomous systems is still rather underresearched. In this paper we have investigated the application of natural language interaction with autonomic systems, focusing especially on log parsing and analysis.

Our experiences with the industrial usage of a prototype implementation, the NL4AC system, have shown that . . . *

* depends on lab report

More work remains to be done. Besides improvements in the language recognition techniques, other interesting opportunities concern the way occurrence of patterns of interest is reported back to the user. Currently, correlations are shown as groups of arrows in a sequence diagram. This diagrammatic representation may be good to provide an overview, but specific patterns may be better reported in natural language, in the same style as the sentences used as input. A clear, short textual form such as

An error about "web server" occurred and margot is down.

is easier to interpret even for non-specialized personnel, and moreover can be effectively transmitted (e.g., paged) to an off-site operator. We believe that improvements in user interfaces for autonomic system will significantly increase the penetration of autonomic technology in a wider range of environments.

References

- [1] Vincenzo Ambriola and Vincenzo Gervasi. NL4AC: Primo rapporto tecnico. Technical report, May 2005. (in Italian).
- [2] Vincenzo Ambriola and Vincenzo Gervasi. On the systematic analysis of natural language requirements with Circe. *Automated Software Engineering*, 2005. (to appear).
- [3] Vincenzo Gervasi. *Environment Support for Requirements Writing and Analysis*. PhD thesis, University of Pisa, March 2000.
- [4] Vincenzo Gervasi. The Cico domain-based parser. Technical Report TR-01-25, University of Pisa, Dipartimento di Informatica, November 2001.

- [5] Vincenzo Gervasi. NL4AC: Eclipse plug-ins technical description. Technical report, January 2006.
- [6] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, January 2003.

*

* We need clearance from IBM to refer the two Technical Reports.

Contents

1	Introduction	1
2	The current scenario	1
2.1	Autonomic problems	2
2.2	Current technology	3
2.3	Current process	3
3	The NL4AC approach	5
3.1	Features description	6
3.2	Architecture description	6
3.2.1	Logical components	7
3.2.2	NL4AC/Eclipse interface	7
3.2.3	Delegating Log Correlator	8
3.2.4	Packaging	8
3.3	Operations	9
3.3.1	Creating a new correlation file	9
3.3.2	Editing a .corr file	10
3.3.3	Saving a .corr file	11
3.3.4	Testing a correlation	13
3.4	Language	14
3.4.1	An example	15
3.5	Customizability	15
4	Industrial validation	17
5	Related work	17
6	Conclusions and future work	17