NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 — "Semantic-based knowledge and content systems"

D2.3.5 C-ODO plugin updates and extensions

Deliverable Co-ordinator: **Alessandro Adamou**

Deliverable Co-ordinating Institution: **CNR**

Other Authors:

This document presents an overview of new functionalities and improvements on existing ones for the Kali-ma plugin, which exploits the C-ODO Light ontology for enhancing user interaction with the NeOn Toolkit.

This upgrade focused on enhancing the overall interaction experience, and on introducing features targeted at supporting both collaboration among ontology engineers and interoperability between other plugins and with Kali-ma itself.

Since the new version of the plugin now provides development tools for other contributors to extend their plugins with interoperability features, these tools are both described on the operational level and documented for NeOn Toolkit developers.

Document Identifier:	NEON/2009/D2.3.5/v1.2	Date due:	February 8, 2010
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	February 2, 2010
Project start date	March 1, 2006	Version:	v1.2
Project duration:	4 years	State:	Final
		Distribution:	Public

NeOn Consortium

This document is part of the NeOn research project funded by the IST Programme of the Commission of the European Communities by the grant number IST-2005-027595. The following partners are involved in the project:

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, e.motta}@open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB Englerstrasse 11 D-76128 Karlsruhe, Germany Contact person: Peter Haase E-mail address: pha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.ump.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Jesús Contreras E-mail address: jcontreras@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SL–1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 665 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier, France Contact person: Jérôme Euzenat E-mail address: jerome.euzenat@inrialpes.fr	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield, United Kingdom Contact person: Hamish Cunningham E-mail address: hamish@dcs.shef.ac.uk
Universität Kolenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Marino della Battaglia 44 – 00185 Roma-Lazio Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Food and Agriculture Organization of the United Nations (FAO)** Viale delle Terme di Caracalla 00100 Rome Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@fao.org
Atos Origin S.A. (ATOS) Calle de Albarracín, 25 28037 Madrid Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com	**Laboratorios KIN, S.A. (KIN)** C/Ciudad de Granada, 123 08018 Barcelona Spain Contact person: Antonio López E-mail address: alopez@kin.es

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to the writing of this document or its parts:

- CNR
- FAO
- JSI
- ONTO
- UKARL
- UKO-LD
- UPM
- USFD

Change Log

Version	Date	Amended by	Changes
0.1	05-01-2010	Alessandro Adamou	TOC and early layout
0.2	09-01-2010	Alessandro Adamou	Added sections for UI and plugin access upgrades, with figures
0.4	12-01-2010	Alessandro Adamou	Finalized UI section, added interoperability API section
0.5	13-01-2010	Alessandro Adamou	Added introduction, updated whole interoperability section
0.6	15-01-2010	Alessandro Adamou	Finalized interoperability section, moved usage section to appendix
0.8	18-01-2010	Alessandro Adamou	Finalized introduction, conclusion and appendix
0.9.5	19-01-2010	Alessandro Adamou	Added collaboration section, overall revision
1.1	02-02-2010	Alessandro Adamou	Applied reviewer comments, added implementation updates to chapter 2, added executive summary
1.2	10-02-2010	Alessandro Adamou	Final QA

Executive Summary

The purpose of this deliverable is to provide an insight on the improvements and additions proposed and implemented for the Kali-ma plugin since its first public release. The document is intended to confirm the outcome of the short-term implementation plan outlined in the “Ongoing Work” chapter of deliverable D2.3.4, which describes the functionalities, motivation and architecture of Kali-ma, the C-ODO plugin. A preliminary review of D2.3.4 is therefore advisable for a better understanding of the grounds which the second version of the tool improves upon.

The current version of the tool garnishes the plugin organization interface with a dynamic graph-like view whose expressiveness is actually enhanced with respect to the original tree-like view that is still available as an option. Third-party tool support has been extended with the ability to run actions or open wizards and help pages provided by the tools themselves. A realtime chat service is being added for user collaboration support, while APIs are now available for allowing other plugins to contribute to the Kali-ma user interface, either by controlling the dashboard or by running tasks from within the dashboard itself. Finally, we implemented a search function that allows users to retrieve up to four different types of data and metadata generated by either the NeOn Toolkit core or selected plugins.

The appendix attached to this document is intended for developers to gain knowledge on how their plugins can be extended to support Kali-ma for integration and interoperability, in addition to the OWL-based method that still holds from version 1.0. This conclusive section describes both APIs for dashboard control and plugin interoperability, and comes with examples and suggested practices for avoiding strict dependencies between third-party plugins and Kali-ma.

Contents

List of Figures

Chapter 1

Introduction

In Deliverable D2.3.4 [AGP09], the C-ODO plugin for the NeOn Toolkit, named **Kali-ma**, was introduced in its conceptual, functional and infrastructural aspects. The document included an analysis of recurring metaphors in the interaction flow between ontology engineers and the Eclipse environment, which is the basis for the NeOn Toolkit (shortly NTK or “the Toolkit”). Some implications of employing these metaphors were outlined, and the setbacks deriving from this usage were assessed in the NTK context. Along with this analysis came our proposal, in the form of the Kali-ma plugin, for an interaction environment that can help users in overcoming these hurdles, which mainly derive from an unmanaged and potentially inconsistent model of the NeOn Toolkit platform as it emerges from the ways to access its functionalities. Our document included a real-world application scenario inspired from the pharmaceutical case study in WP8 (cf. [CL08]), plus an evaluation of preliminary user feedback obtained during the development phase of the tool.

The first release of the Kali-ma plugin was centered on introducing ontology developers and NTK users in general to an alternative, widget-based user interface view on the platform. From said view, the Toolkit itself is interpreted as a functionality provider, where these functionalities can all be accessed through similar interaction paths of equal length. To demonstrate the benefits deriving from this user interface, we included an early implementation of features such as plugin access methods and organization based on semantics.

While the features implemented in version 1 were still far from our final goal, this early release was a necessary premise for testing how the interaction paradigm adopted by Kali-ma would be well-received. Moreover, we took advantage of this version in order to assess the perception of our approach as a valid alternative to standard Eclipse mechanisms, which we argued to be at high risk of being inconsistent by a design-centered model of the platform, and even going as far as to (unwittingly) conceal some functionalities from users. Related arguments and an analysis on the feedback received were discussed in [AGP09].

With this first release in place, paving the way for extending the range of our approach, we are proceeding to plug more functionalities into the Kali-ma tool, overhauling existing ones in the meantime, all in accordance with the chosen interaction paradigm. Our goals for this updated version are: *(i)* to strengthen the interactive capabilities of the Kali-ma dashboard system for end-users; *(ii)* to promote the vision of the dashboard system as a host of services, which plugin providers can deploy by simply wrapping the functionalities already implemented; *(iii)* to introduce functions by which Kali-ma can natively act as a contributor to the collaborative engineering of Web ontologies; finally, *(iv)* to make users aware of a NeOn Toolkit workspace or project as way more than simple ontology stores, but as broad, searchable repositories of knowledge and meta-knowledge, that can be explored *even in the absence of the plugin(s) that generated it*.

The rest of this document will be structured as follows. Chapter 2 is the core of the entire deliverable, as it documents, both for end-users and, to some extent, contributing developers, the whole set of upgrades implemented upon Kali-ma since version 1. Final remarks and further development strategies are discussed in Chapter 3. To conclude, since some new features involve the use of application programming interfaces (APIs) for plugin developers, a usage guide to such APIs is also provided as an appendix.

Chapter 2

Kali-ma updates and extensions

This chapter is structured in sections where the new features are grouped by significant categories on the functional level: Section 2.1 concentrates on user interface improvements, with particular focus on the revamped Organizer widget. Section 2.2 outlines those features aimed at enhancing the Kali-ma plugin as a collaboration-oriented tool. Section 2.3 describes in what additional ways Kali-ma is now able to allow users to interact with other NeOn Toolkit plugins, besides the basic methods already present in the first version. Section 2.4 introduces new features through which Kali-ma can facilitate the mutual co-operation of third-party plugins with each other as well with Kali-ma itself.

2.1 User interface enhancements

In addition to improving the Kali-ma tool on the functional level, development kept focusing on the maintenance of the user interaction aspect, taking up from the intuitions and feedback that emerged during the first stable development cycle. The release of a second version offered us an opportunity to renovate the most significant parts of the Kali-ma dashboard so as to render it semantically closer to the "dial and switch housing" metaphor that our Graphical User Interface GUI is named after.

Most of our GUI enhancement efforts were concentrated on providing a more powerful way to render the space of NeOn Toolkit plugins in terms of their categorization rules. As with Kali-ma v1.0, this release still offers a choice of three classification criteria for ontology design tools. While we refer to the previous WP2 deliverable [AGP09] for a thorough outline of these criteria, it is useful to recall them here as being:

1. Ontology engineering **activities** according to the NeOn methodological guidelines (cf. deliverable [SFBd$^+$09]).
2. Arbitrarily defined **design functionalities** implemented by tools (usually defined by plugin providers themselves).
3. Specialized functionalities defining **design aspects** of networked ontology engineering (proposed by this workpackage in deliverable 2.3.2 [PMP$^+$09]).

The Kali-ma dashboard features a widget, called **C-ODO organizer**, whose function is to present the user with an overview of existing, or installed, NeOn Toolkit plugins according to the preferred criterion. In the previous version, this overview was only available in the form of a two-level tree structure, with the leaves being NTK plugins and their parents being the categories that best describe these plugins from a functional standpoint.

This graphical representation form, albeit sufficiently functional, tended to fall short of aesthetic appeal, but most of all, suffered from an inherent limitation of acyclic hierarchical structures, i.e. node redundancy. Regardless of the classification criterion selected by the end user, some ontology design tools will most likely pertain to multiple categories. For example, the *Cicero* plugin supports both the *Ontology assessment* and

Ontology evaluation activities, while the *XD Tools* plugin is classified as covering both the *Design patterns* and *Reuse and reengineering* design aspects. The most obvious way of representing such multiple categorization as a tree is to replicate the nodes that represent plugins, which has a potential for confusing end users.

To make up for this downside, the second release of Kali-ma includes a further rendering mechanism, while still retaining the old tree view for the benefit of users who might be accustomed to it. This interface element was dubbed the **wheel view**, as a reference to the radial layout strategy it adopts when no tools or categories are selected. Figure 2.1 shows an example of a wheel view as shown on startup, in the case where design aspect coverage is chosen as a criterion. Initially, tool categories are laid out in a radial fashion and no tools are displayed. When the user selects a category by clicking on its node, the plugins for that category appear, and the view is rearranged as a bipartite graph: tool categories make up one partition, while NeOn Toolkit plugins make up the other. Connections between partitions indicate which categories a plugin belongs to, i.e. which design functionalities are implemented, which NeOn methodology activities are supported, or which design aspects are covered by that plugin. If the user clicks on an empty region of the wheel view, the graph is arranged back to its original state, displaying only categories in a radial fashion.

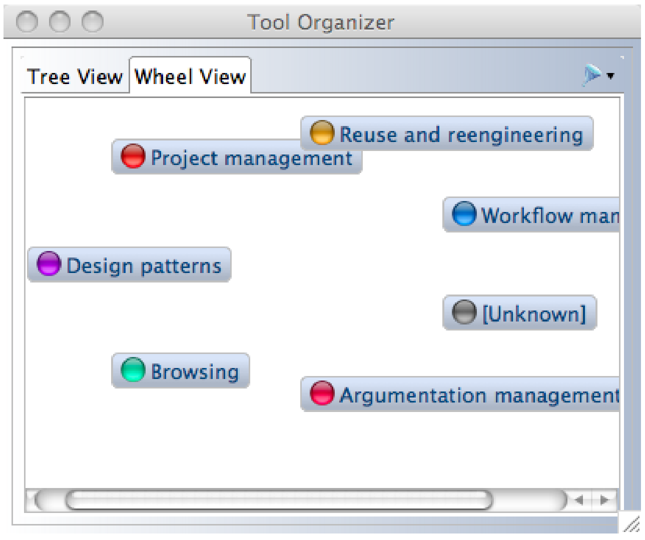


Figure 2.1: C-ODO organizer widget showing known design aspect in its wheel view on a fresh start.

As with the tree view, a plugin widget can be opened by simply double-clicking on a node representing an installed plugin. Also, the wheel view sports the same context-sensitive features as those implemented in version 1.0 for the tree view. The *Helper widget* will display natural language information about a selected plugin or category, as extracted from the corresponding ontologies describing it. The wheel can be filtered in order to show all the NeOn Toolkit plugins that are not installed on the running system, or even those ontology design tools that are not NeOn Toolkit plugins at all but are, for example, Web applications or standalone desktop tools. Additionally, it is now also possible to filter out all the categories for which no ontology design tool is known to exist at a given time. Note that this is independent of the outcome of any other filters applied to the view, as categories for which plugins exist but are filtered out will appear anyway. This additional filter was provided in order to minimize unnecessary overcrowding of the organizer view, with special attention on activities from the NeOn methodology, which currently amount to over 50 items, yet at least half of them are awaiting plugins to be registered. As for the remaining criteria, since design functionalities are instantiated by plugin providers themselves, it is highly unlikely for any of them to be left blank, while we have now been able to classify at least one plugin for each of the six design aspects defined.

The alternative graphical view that was implemented for this version is more significant than a simple user interface embellishment exercise. While the wheel view may not be as compact by sheer geometrical terms, it does provide more information than the standard tree view does. An example of how such greater expressiveness is accomplished can be seen in Figure 2.2. Here, a comparison is drawn between the old tree view (left) and the new wheel view (right) running on the same NeOn Toolkit instance, with the same set of tools classified by their ontology design aspects. In the tree view, the nodes for both the ‘Design patterns’ and ‘Reuse and reengineering’ aspects were expanded, and it can be noticed how the child nodes of the former are but a subset of the child nodes of the latter. By the rules that describe these aspects and the corresponding plugins in their OWL manifests, this is perfectly normal. However, this leads to duplicate nodes for the plugins in common, which can be misleading without even allowing the user to know more about these plugins, than strictly the information requested. In the wheel view to the right, only the ‘Reuse and reengineering’ aspect was queried; however, the connections between nodes from the two partitions indicate which of the reuse and reengineering tools also belong not only to the ‘Design patterns’ aspect, but to *every other* aspect in the taxonomy. For example, the user is told that the *Reasoner* and *Ontology Customization* plugins also cover the ‘Argumentation management’ aspect, while the Reasoner plugin is also suitable for ‘Workflow management’ (which makes sense, due to the high versatility of DL reasoning capabilities, as proved by the Kali-ma plugin itself). All this without any need for duplicating nodes whatsoever.

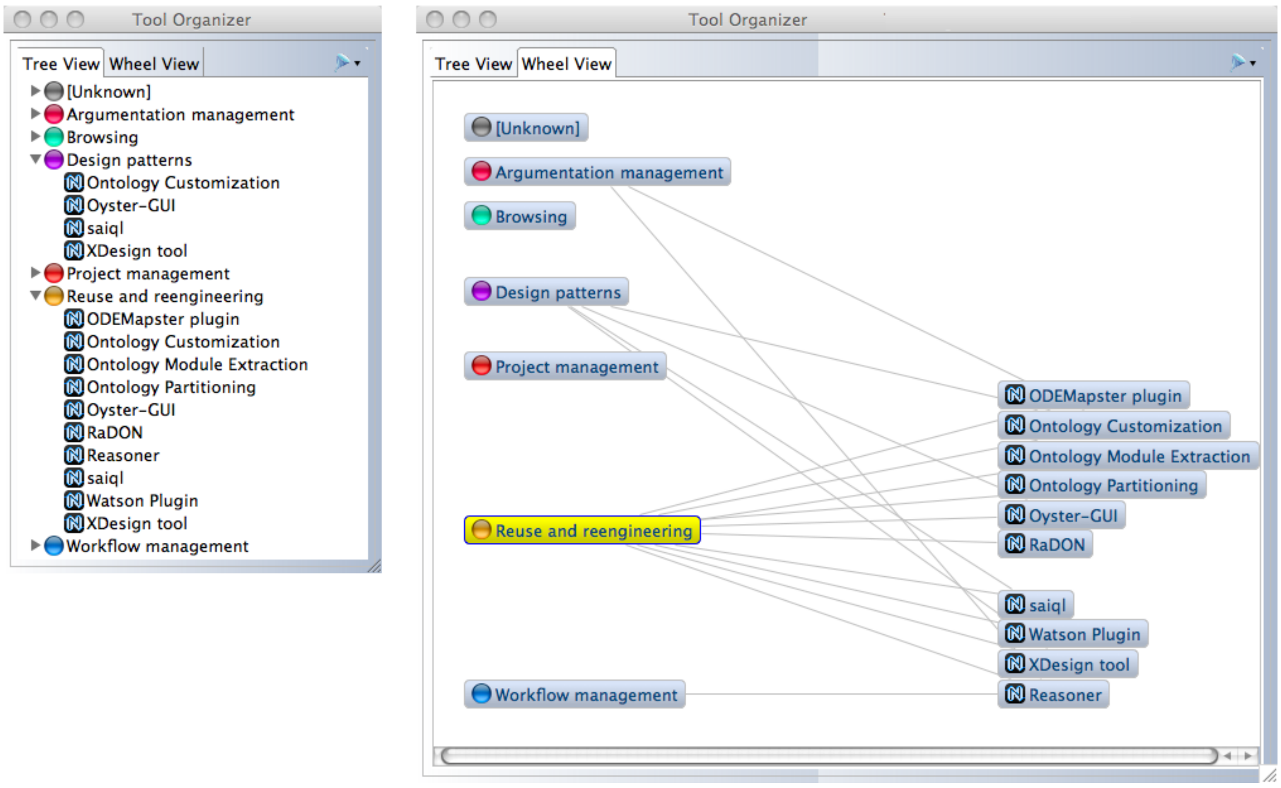


Figure 2.2: Comparison of the two organizer views displaying NeOn Toolkit plugins for the ‘Design patterns’ and ‘Reuse and reengineering’ categories. Notice how the wheel view eliminates all redundancy and displays more design aspects supported by those tools, than the ones queried by the user.

The wheel view was entirely developed using the Zest library[1]. Zest is a visualization toolkit that offers support for displaying read-only graphs (i.e. not used for graphical editors but mostly for displaying information) and customizing layout policies. Also, the Zest framework has the undeniable advantage of using the JFace UI toolkit [SHNM04], which is the same used in the tree view for decoupling the Kali-ma datamodel and user interface, hence implementing a Model-View-Controller architecture[Ree79], as well as supporting filters like the ones mentioned earlier in this section. Although seamlessly integrated with Eclipse, the Zest library does not come as part of the NeOn Toolkit base distribution, but it relies upon visualization components, such as Draw2D and the Graphical Editing Framework (GEF)[2], that are included in the extended version of the NeOn Toolkit. Therefore, it is perfectly feasible for the Zest library to be shipped either as part of the Kali-ma plugin (for the extended NTK) or a visualization library plugin, which is already available on the update site for the

[1]`http://www.eclipse.org/gef/zest/`
[2]`http://www.eclipse.org/gef/`

basic NeOn Toolkit 2.3.

In addition to implementing the wheel view, we kept following up on the feedback received during the evaluation session that was held halfway through the development phase of Kali-ma v1.0 (cf. [AGP09], Section "User evaluation"). In response to the widget overcrowding issue that was brought about by practitioners, and to recommendations received during the internal review phase, we enhanced the **Dock widget**, which maintained placeholders for plugin widgets hidden from view. In its latest stage, this widget now features a row of toolbar buttons which, upon selection, toggle the visibility of non-essential native widgets, such as the aforementioned Helper or the **Profile manager**.

2.2 Collaboration support

Being C-ODO Light a networked ontology aimed at identifying and leveraging those aspects of ontology design that can, or need to be either shared or performed collaboratively, it was felt appropriate to equip Kali-ma with some sort of "communication channel" in order to favor the process. As thoroughly described in [DEB$^+$08], which is reprised later in this document (cf. Section 2.4.1), the Cicero Wiki and plugin provide valuable support to the evaluation of an ontology through discussion on its single components. It has to be noted, however, that Cicero was mainly designed to support specific activities in the lifecycle management of ontologies, such as argumentation, evaluation, assessment and documentation.

As with the definition of ontology *design aspects* (cf. [AGP09]), our goal is to have a collaborative feature that does not target any specific phase in the ontology lifecycle management and is, in a way, transversal to the whole methodology. Also, it has to offer an alternative to the fully asynchronous and persistent nature of a Wiki, whose rigorous structure may have a psychological impact on end-users as to get them to deem it unfit for quick, transient discussions on any matter or activity in an ontology engineering process.

A possible solution to this issue was identified as simple instant-messaging, or *chat* support. Examples of this functionality have already been featured in the state of the art, as was the chat service available in Collaborative Protégé [TNTM08]. In that case, however, the chat service was part of an infrastructure where entire project data were being shared, thus implying that users had to "connect to a project" in order to share or discuss its data. On the other hand, the NeOn Toolkit uses a local, fully decentralized project model, which leaves users at liberty to opt for their own sharing mechanisms, such as SVN versioning systems and alike. In the absence of a shared infrastructure for packet relaying and broadcasting within NeOn, we opted for what was deemed the most promising off-the-shelf legacy technology, i.e. Google Wave[3].

Along with the evolution and ongoing development of the Wave technology, we are maintaining a special native widget, temporarily named **Wave widget**, which is mainly a lightweight browser window whose start page embeds a Wave object. A Wave is the sum of a collaborative web-based service and a realtime communications protocol allegedly designed to merge email, instant messaging and wikis. The federated communications protocol has the string advantage of extending the *de facto* standard Extensible Messaging and Presence Protocol (XMPP). The default Web application for displaying Waves has proved to emulate realtime chat in an adequately efficient manner. Additionally, developer tools for customizing Wave gadgets and adding robot participants are already available, while its communications protocol was disclosed in July 2009 as an open protocol.

Currently, Wave implementations are at a highly experimental stage and a number of limitations, such as the restriction of Google Inc. as a unique Wave provider and the incompatibility of embedded Waves with Microsoft Internet Explorer, are being addressed. Additional restrictions, such as invite-based participation both for end-users and developers, are scheduled to be relaxed by the end of the ongoing preview phase. We are closely following the advancement of these technologies and tracking down the way this evolution unfolds to the advantage of the Kali-ma instant messaging widget. Parallel to this process, the possible development of a simple XMPP-based fallback solution for the benefit of all NeOn technologies, should Wave technologies fail to be on par with the expectations mentioned above, is being investigated.

[3]`http://wave.google.com`

2.3 Plugin access method support

In compliance with our development plan as roughly sketched in the “Ongoing work” section of [AGP09], support for additional interaction modes provided by NeOn Toolkit plugins has been implemented. As NeOn Toolkit users and QA testers, we came to realize how plugins, both released and in development, offered so great a variety of ways to contribute to the NeOn Toolkit UI, that it definitely needed broader coverage. In other words, Kali-ma users needed a more precise interpretation of the notion of “opening a plugin”, one that went beyond the mere action of displaying a composite panel, or “Perspective”, which is provided by roughly one third of available plugins. An early example of this effort was already present in version 1, which supported “New wizards”, multi-page guides for creating new resources from scratch, an instance being the gOntt wizard for scheduling an ontology development plan in a guided way.

The discovery of access methods for each plugin is strictly related to the Eclipse implementation and does not rely on the OWL descriptions of that plugin. Kali-ma becomes aware of supported access methods at runtime, by scanning the *extension registry* of the NeOn Toolkit, which contains information about all supported extension points, as well as the Java classes that implement the corresponding extensions. In addition to supporting *Perspectives*, *Views* and *New wizards*, the second public release of Kali-ma supports the following access methods:

1. *Actions* are placeholders for commands that can perform virtually any kind of operation. In an Eclipse application, actions are attached to buttons, toolbars or menu items. Their run method, which may include opening views or wizards, is called whenever the end user clicks on a menu item or button. Plugins such as Cyc, Oyster-GUI, SearchPoint and the GATE services contribute to the NeOn Toolkit UI by means of actions. Kali-ma itself is activated through an action.

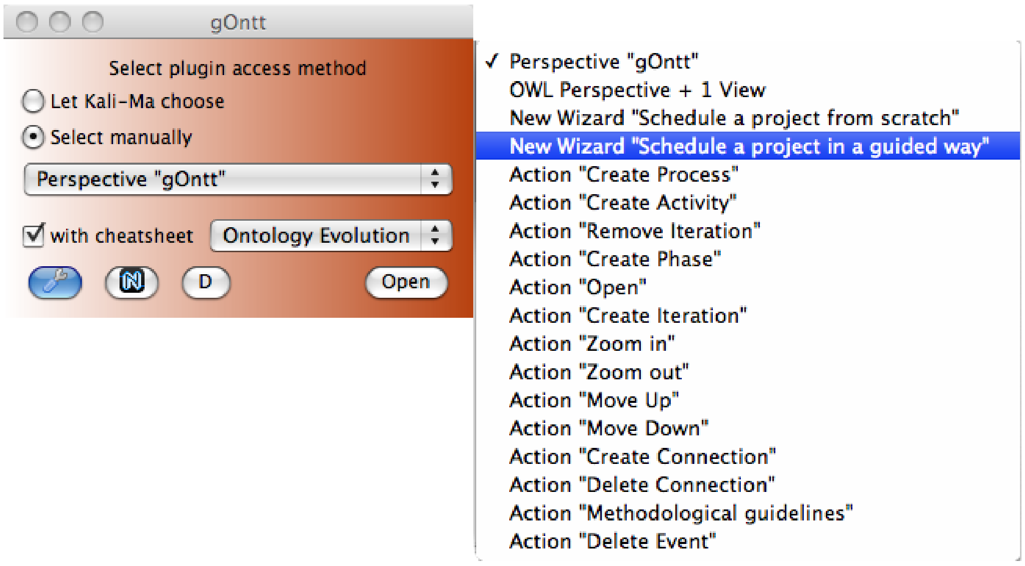
2. *Cheat Sheets* are interactive help pages that guide users through some ontology development processes. Each cheat sheet lists the sequence of steps required to help achieve a certain goal. Cheat sheets can launch required tools automatically, and require users to perform manual steps as they progress through the task. A cheat sheet opens as a view, and only one cheat sheet is open and active at any time. Most NTK plugins come with at least one cheat sheet, as it is good practice to provide one for each implemented use case. As Kali-ma does not rely on workbench views for its user interface, it uses the Helper widget described in [AGP09] instead of a cheat sheet.

3. *Export Wizards* are paged dialogs for guiding the user through a process of saving a given workspace resource or converting it across representation formats. The OWLDoc plugin for automatically generating Javadoc-like documentation of ontologies is a good example of export wizard, especially because it does not require the source ontology to be selected in the workspace before launching the wizard: it can be selected from within the first wizard page.

When multiple entry points are available for a single plugin, Kali-ma adopts the following strategy in order to launch a plugin, unless manually overridden according to widget-based settings: if a Perspective is present, it is opened first; otherwise, Kali-ma falls back to, in order, New Wizards, Actions and Export Wizards. If none of the above are available, Kali-ma will switch back to the NeOn Toolkit workbench and show its standard OWL Perspective, adding to it as many Views as are defined by that plugin. Users have an option to accompany the chosen Perspective or View with a Cheat Sheet selected from the ones made available by that plugin.

Summing up the access methods previously supported with the ones added by this new version, Kali-ma is now able to provide meaningful entry points to the vast majority of plugins developed for the NeOn Toolkit platform by project partners. As a proof-by-example, Figure 2.3 shows how dramatically the list of access method choices has increased for the gOntt plugin between versions: a similar figure in [AGP09] displayed only the first three options (the fourth not being available due to the gOntt plugin providing only one wizard at the time). Additionally, the list items now bear a more human-readable name than the previous unique namespace identifier. These labels are extracted straight from the respective extension in the plugin’s manifest file.

Note that the execution of actions can depend on certain preconditions: in the case of gOntt, actions like “Move Up”, “Zoom In” or “Delete Event” assume a gOntt schedule to be already loaded in its perspective. Because action management occurs directly in the plugin code, it is not possible for Kali-ma to distinguish between actions that will or will not result in an effect. On the contrary, cheat sheets are mostly guaranteed to be openable by standard means.


gOntt
D
Open
✓ Perspective "gOntt"
OWL Perspective + 1 View
New Wizard "Schedule a project from scratch"
New Wizard "Schedule a project in a guided way"
Action "Create Process"
Action "Create Activity"
Action "Remove Iteration"
Action "Create Phase"
Action "Open"
Action "Create Iteration"
Action "Zoom in"
Action "Zoom out"
Action "Move Up"
Action "Move Down"
Action "Create Connection"
Action "Delete Connection"
Action "Methodological guidelines"
Action "Delete Event"

Select plugin access method

Let Kali-Ma choose [✓ Select manually]

Perspective "gOntt"

[✓] with cheatsheet Ontology Evolution

Figure 2.3: List of access methods supported by Kali-ma for the gOntt plugin. All the actions provided by gOntt are now part of the list, although some actions may require a gOntt plan to be already loaded, or the gOntt Perspective to be already shown. A cheat sheet can be optionally selected from a second drop-down list.

Although a possible implementation would have been similar to the one for Export wizards, hence very straightforward, we have opted for not supporting the *Import wizard* access method. An Import wizard is a paged dialog that allows the user to bring in a resource from some context (e.g. the local filesystem, the Web or a DBMS) and make it available as a workspace resource. Currently, this access method is the means by which the NeOn Toolkit core is able to load OWL ontologies from the Web or the filesystem, but it is not being used by any known plugins, with the exception of the now obsolete Robust OWL Import plugin. Additionally, because most resources handled by the NeOn Toolkit or its plugins belong to ontology development projects, it is highly likely that the user will need the tree view of the Navigator in order to select one such project, or resource contained therein, before launching an Import wizard.

2.4 Plugin interoperability

The first public release of Kali-ma was focused on presenting an alternative view of the NeOn Toolkit platform as a functionality provider, and on offering seamless access to these functionalities. It can be argued that such operational flow is basically uni-directional, as Kali-ma’s job was to bridge the NeOn Toolkit and its plugins by keeping them completely agnostic of the user-centered mediation that Kali-ma purveyed. We name this feature “plugin integration”, and its most evident example is access method support as described in Section 2.3 of this deliverable.

In parallel with enhanced collaboration support among users, this new release moves several steps forward from integration towards interoperability between applications, provided by NeOn Toolkit contributors by means of plugins. With the first milestone, initial support for *metalevel integration* was added. We have made the Kali-ma tool aware of specific types of metadata that some plugins store into the user workspace at runtime, and made the most significant ones available for user query (cf. Section 2.4.1). As a demonstration of how the results of a metadata search can be leveraged, we introduced a context-sensitive mechanism that allows plugins to consume the resulting knowledge objects as an input to their own design functionalities (or a subset thereof) where needed, and present new knowledge objects to the system, for other plugins in turn to consume. In addition to providing this method for allowing other plugins to interoperate, the need for a specific integration mechanism with the gOntt plugin hinted at offering another interoperability feature which, in fact, allows any plugin other than Kali-ma to manipulate the dashboard programmatically, in order to facilitate the execution of scheduled activities or simply maintain a coherent mental model of the NTK platform throughout the execution of multiple tools.

Plugin integration is achieved by scanning a runtime configuration of the NeOn Toolkit and matching its entries against the knowledge described in a semantic subsystem, which aggregates and reasons upon OWL modules describing known ontology engineering tools from the Semantic Web. The procedure and external tools to support this mechanism are thoroughly described in our previous deliverable (cf. [AGP09] Chapter 3, “OWL plugin description management”). Therefore, integration does not require any kind of intervention on the source code or configuration files from developers. On the contrary, to achieve interoperability as defined in the previous paragraph, with the exception of metalavel integration (which is, indeed, a form of integration, although it is being used for interoperability purposes), it is necessary for plugin developers to include a few, simple extensions to their code. In principle, these extensions should not alter the functionalities already implemented by plugins, even though developers may deliberately choose to do otherwise. This document will also provide guidance on using the APIs exposed by Kali-ma for implementing these extensions.

2.4.1 Metalevel integration

The process of upgrading the NeOn Toolkit and its plugins, as well as migrating them to a new OWL data-model and Eclipse platform, gave us an opportunity to analyse, source code in hand, some behavioral patterns adopted by the NTK core and plugins for managing data and metadata, including but not limited to ontologies and their entities, during a typical round of execution.

There are no set standards or rules for storing metadata generated by plugins in an Eclipse environment. However, the Eclipse Rich Client Platform provides facilities for doing so, which imply a handful of best practices for distinguishing which types of metadata should or should not “survive” when either a project or an entire workspace is ported to other instances of the environment. The rule of thumb for metadata storage is to use the plugin state location[4] for non-sensitive information that does not pertain to any specific project and can be rebuilt from existing data without any consistent loss, and the project directory for project-specific metadata. For instance, Kali-ma stores its reasoning cache and merged plugin description ontologies in its own state location, as they are cross-project metadata that may apply to the entire workspace. We may want to change this behaviour in the future, if we decide to support a tight linkage to ontology projects within Kali-ma.

Based on these best practices, we have conducted a preliminary analysis of the metadata stored both the NeOn Toolkit 2.3 pre-release and the first batch of new or migrated plugins for it. The aim was to determine which of these tools actually do store these metadata locally, and which of these can be considered meaningful to either end-users or other plugins that can accept them as input. For this reason, no prior assumption was made as to which data types to look for, nor which plugins to monitor for the generation of these data, which should be guaranteed to be persistent even in environments where the originating tool is not installed. These data would then be made available for retrieval by a new native widget for searching them and making them available, along with OWL data, to the context-sensitive Kali-ma dashboard system. Our findings summarize as follows.

The NeOn Toolkit core basically uses the state locations of its components in order to store simple UI-related or environment-related metadata, such as user preference values, the positions and sizes of views, the

[4]The state location of a plugin is the `$WORKSPACE/.metadata/.plugins/$PLUGIN_ID` directory, where `WORKSPACE` is the directory where projects are stored and `PLUGIN_ID` is the unique identifier of the plugin (e.g. `it.cnr.istc.stlab.kalima`).

perspectives that were open before exiting, etc., while the metadata root directory is used for logging errors and maintaining internal command traces. As for project-related metadata, the NTK uses an XML file to maintain the set of ontologies to be managed by the core datamodel plugin, along with rendering options such as displaying imports or syntax information. Excluding all these metadata, which were deemed of little to no importance to end users or applications, we concluded that, in the case of the basic Toolkit, ontologies themselves are the real source of relevant knowledge worth searching for.

Coherently with our expectations, the pool of NeOn Toolkit plugins proved to be a valuable source of meta-knowledge to be presented, consumed and searched for. However, by Eclipse design principles, developers are left at liberty to decide the amount, locations and formats for storing their own metadata, which means that no unique strategy can be assumed for all NTK plugins. Therefore, it was necessary for us to examine any stored information on a case-by-case basis, and implement adapters for each data type that would be deemed relevant enough to be searched for.

It was then established that plugins other than the NeOn Toolkit core tend to generate project-specific metadata that are stored within project directories in a variety of ways. For example, the *Label Translator* plugin does not store translated RDFS labels back to the ontologies where labelled entities are asserted, but maintains its own ontologies, containing linguistic information based on the OWL version of the Linguistic Information Repository model (LIR)[5], in a project subdirectory called `linguistic`. The *ODEMapster* plugin adopts a similar pattern for storing R2O mappings between databases and ontologies [BOCGP04], except that it uses its own XML Schemas (XSD) for encoding mappings between properties and fields in database schemas, mapping containers and database connection coordinates. Three such XML files are stored for each ontology within the project, using a subdirectory with path `$PROJECT/R2O/$ONTOLOGY_ID`, where `PROJECT` is the full path of an ontology development project, and `$ONTOLOGY_ID` denotes the shorthand identifier (i.e. base URI local name or fragment) of the target ontology. As a conclusive example, the *Cicero* plugin [DEB+08] stores all ontology argumentation sessions remotely in its own MediaWiki[6], but it does maintain these argumentation sessions aligned with ontology entities stored locally by applying its own annotation properties to the local source ontologies. Other plugins opt to rely upon external resources or volatile memory for storing metadata, or not to generate metadata at all.

This premise outlined the motivations and rationale that were necessary in order to introduce the *metadata search* facility provided through a new native widget in the Kali-ma dashboard, quite simply named the **search widget**. Thanks to this additional UI element, NeOn Toolkit users are able to comb an entire workspace or project for data or metadata that conform to the type specifications requested and supported by the tool. Search results are provided in realtime, at every keystroke in the keyword text field, through standard content assistance features provided by Eclipse and JFace.

The available search facets allow users to restrict the search scope to one ontology development project and a selection of data and metadata types to search for. With this release, the Kali-ma search feature supports the following types:

1. **OWL entity names**, i.e. classes, datatype properties, object properties, annotation properties, dataypes and individuals.

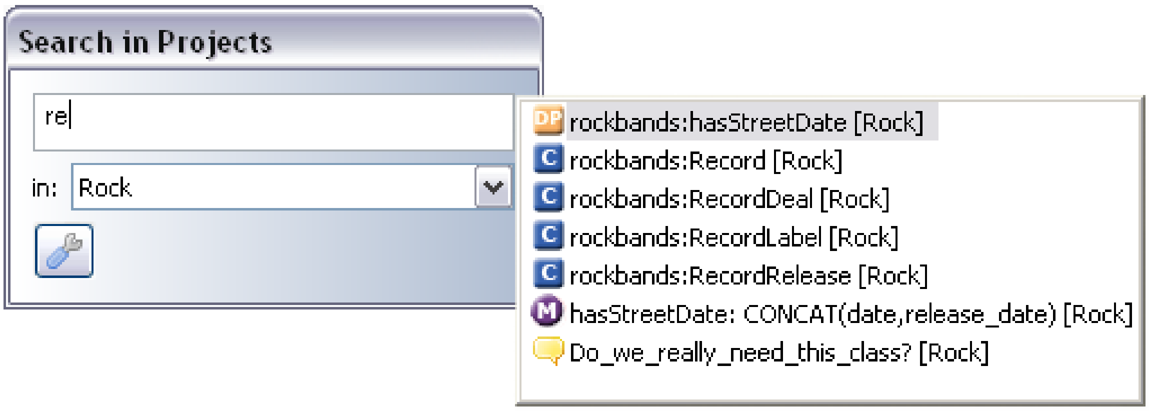
2. **R2O mappings** between ontologies and database schemas. The search scope includes both relational database entity names (schemas, tables, attributes) and involved OWL entity names. We do not search within connection data, e.g. in order to obtain passwords.

3. **Cicero argumentation sessions** by their URIs. The actual session content is stored in the Cicero Wiki and lies outside the search scope.

4. **Linguistic information** based on the LIR ontology and generated, both automatically and by user interaction, through the Label Translator plugin. Typically, it is possible to retrieve ontology entities and associated lexical entries by typing in (a portion of) a translated label.

[5] `http://gate.ac.uk/projects/neon/lir.owl`
[6] `http://cicero.uni-koblenz.de/wiki/index.php/Main_Page`

Figure 2.4 exemplifies the behaviour of the Search widget, two keystrokes since having begun typing the word “record”, after restricting the search scope to an ontology project called “Rock”. In this example, the user has not customized the datatype facet (available by clicking on the wrench icon), therefore the system searches for all supported datatypes and lists their matches in realtime, ordered by datatype. This list, shown on the popup panel to the right, displays seven matches: the `hasStreetDate` datatype property; four classes (`Record`, `RecordDeal`, `RecordLabel` and `RecordRelease`); an R2O mapping that associates the `hasStreetDate` property with the concatenation of the `date` and `release_date` fields[7]; a Cicero argumentation issue, whose URI ends with `Do_we_really_need_this_class?`, associated with the `RecordRelease` class[8].



rockbands:hasStreetDate [Rock]
rockbands:Record [Rock]
rockbands:RecordDeal [Rock]
rockbands:RecordLabel [Rock]
rockbands:RecordRelease [Rock]
hasStreetDate: CONCAT(date,release_date) [Rock]
Do_we_really_need_this_class? [Rock]

Figure 2.4: The Search widget displaying results matching the “re” string within the “Rock” project. Matched results include, from top to bottom: one datatype property, four classes, one R2O mapping and one argumentation session. The wrench icon in the bottom-left corner allows the user to set restrictions on (meta)datatypes to search for.

When browsing realtime search results, it is possible to select a single item in order to complete the input field and trigger the presentation of this item to the dashboard system, for any compatible listening widget to consume as an input datum. This functionality is detailed in the next section and documented in Appendix A.

Note that, for the aforementioned search facets to be available, there is no need for users to install plugins that support or generate them, except for the base NeOn Toolkit itself. As metadata analysis was performed on a case-by-case basis, so was the implementation of the corresponding search providers. Adapters were written ad-hoc for each type group in the list, and adding more as more datatypes as supported will be a fairly easy task. While the adapter for OWL elements is essentially a wrapper of the OWL Search facility of the NeOn Toolkit, the others have no dependency on Cicero or ODEMapster or other third-party plugins.

2.4.2 Interoperability API

The Kali-ma tool makes provision for two distinct methods of allowing *programmatic interoperability*, which is achieved through direct intervention on the plugin code. As documented later in this document, the two methods cover separate interoperability aspects and can be implemented independently. They are:

1. External *Dashboard control*
2. *Computational task execution* within widgets

Dashboard control is an extremely simple API for bridging specifically Kali-ma with another plugin: it is not intended for two plugins other than Kali-ma to interoperate. The role of this API is to allow any plugin to

[7]The database schema used in this example was taken from a local MySQL database about rock band discographies.

[8]The issue starter claims that this class is an overkill since the `hasStreetDate` property would suffice, obviously ignoring that a record album may be released with different dates and record labels in different countries, thus requiring an N-ary relationship to be defined.

launch a new Kali-ma dashboard with a pre-defined set of plugin widgets. By default, when Kali-ma is started the first time after opening the NeOn Toolkit, a fresh dashboard with only native widgets is shown. According to the standard interaction flow of the tool, the user will access plugins either by opening the corresponding widgets from the C-ODO organizer, or by loading a stored profile from the profile manager, or both. Through the dashboard control API, any plugin can override this behaviour and propose its own set of tools to the user, who in turn will have to either confirm or reject this proposal.

This feature was originally intended to support development of the *gOntt* plugin for project scheduling [GPSFV09], in order to have the two tools integrated on the functional level. The gOntt plugin for the NeOn Toolkit allows project managers to layout ontology development plans, where activities and processes per the NeOn Methodology are arranged in a Gantt-like chart. Activity scheduling per se has no direct association with specific plugins, but gOntt sports its own way of discovering which plugins may serve the purpose of carrying out a certain activity, and this discovery is performed at runtime on the NeOn Toolkit installation where the gOntt plan has been loaded (for reference, see the upcoming project deliverable D5.3.3 due M48). The first step towards gOntt integration was performed during the development of Kali-ma v1.0, when we added support for classifying design tools with respect to the activities supported. This feature was not originally planned, but was found to be essential in order to accommodate gOntt end-users, who could experience disorientation in shifting across two orthogonal conceptual models of the NTK platform, i.e. the NeOn methodology and *codolight*-based design aspects. With the dashboard control API, it is now possible for Kali-ma to assist ontology engineers in performing a development phase or set of planned activities at a given point in their gOntt schedule. Users will be able to pick a selection of plugins that can support the whole phase at once, launch a Kali-ma dashboard with the corresponding plugin widgets (including one for gOntt itself) already opened, and switching across these plugins at will. Having a gOntt widget in the dashboard eases the process of switching back to the gOntt plan in order to advance in the schedule or adjust it. The first example of dashboard control API exploitation is available in gOntt from version 1.3.7, which allows users to launch a Kali-ma dashboard including widgets for all the plugins that are registered for a given activity, or process in a gOntt plan. At the time of writing, development for additional interoperability features is underway.

All of the API methods, described in Appendix A, require that a mnemonic unique identifier be passed along with the list of plugin names. This identifier is used as a proposal for a new profile name, yet it does not automatically trigger a profile save operation, since for any reason the user might be unwilling to accept this set of tools permanently. It is sufficient to click the save button located in the bottom area of the profile management widget in order to have that plugin set stored as a profile with the given name. Then, it will be possible to bind this profile to one or more ontology development projects, as with any other dashboard profile. Additionally, developers have the option to override some user preferences for Kali-ma, by either enforcing a preferred classification criterion (useful for gOntt to maintain the NeOn Methodology model across plugins) or requesting all plugin widgets to be initially docked (useful for a large amount of plugins). Note that dashboard control also works on an existing Kali-ma dashboard, in which case a call to the API will supersede any profile or widget set already open. However, if Kali-ma has already been launched during the session, any attempt to override the classification criterion will only take effect only upon restarting the NeOn Toolkit.

Computational task execution allows developers to expose part of the functionalities implemented by their plugins as services that can be invoked straight from the Kali-ma dashboard, without having to interact with plugins through the NeOn Toolkit workbench. An operation that implements such a functionality is called a *Computational task*, in accordance with the notion of *Computational design task* that is present in the interaction module of C-ODO Light[9] and identifies “any type of design operation (i.e. a functionality) that needs to be performed on a tool”[10].

The dashboard control system implemented in Kali-ma is sensitive to the exposure of knowledge objects, such as OWL ontologies or entities. These knowledge objects can be presented to the system either as a

[9]`http://www.ontologydesignpatterns.org/cpont/codo/codinteraction.owl`

[10]Retrieved as the English RDFS comment for the `ComputationalDesignTask` class.

result of a data-level or metadata-level search performed through the Search widget (cf. Section 2.4.1), or as produced in output by widgets that are representatives for plugins implementing this API. In the latter case, what plugin providers need to do is write a wrapper class that invokes the implementation of a functionality that already exist in their plugins, and returns its output in a form that is convenient for Kali-ma. Details of this procedure for developers are provided in Appendix A.

Implementing a computational task grants Kali-ma a direct entry-point to a functionality implemented by that plugin, but also has a fundamental impact on the user interface for the corresponding dashboard widget. Its main panel will no longer be limited to a natural language description of the tool, but will also feature whatever user interface controls are necessary in order to provide the entry point with the input parameters needed for its function to be invoked. These input parameters are essentially specified by the developer as a list of Java types in the wrapper class. This is due to the fine-grained implementation of computational task invocation, which requires strict type safety, as it needs to use reflection techniques [FF04] in order to match datatypes at a much lower abstraction level than the one available in the codolight-based OWL descriptions of plugins. Given this list of types, Kali-ma employs its own internal heuristics in order to determine which UI controls to display on the widget. The list below outlines some examples of the most common type-control matchings expected:

- a **string** is mapped to a *text field* where the user can input its value;
- an **integer** is mapped to a *spinner*, i.e. a text field that only allows integer values, with arrow buttons for increasing or decreasing its value;
- a **boolean** is mapped to a *checkbox*;
- an **enumeration** is mapped to a *combo box*, which allows a choice of one among all the members of the enumeration thanks to a drop-down menu;
- a **file** is mapped to a *file selector*, which is provided by most operating systems for selecting a resource from the filesystem;
- an **OWL entity** (e.g. a class, individual or property) is not mapped to any visible interface control, but the widget will be automatically put on a listening state in order to get a handle on any such entity that is presented to the system through another widget;

Instantiating these mechanics with a simple yet meaningful example, suppose to have a plugin that is specialized in creating new OWL classes that are related to an existing class by either subsumption, equivalence or disjointness, and automatically adds annotations such as RDFS labels to it. The plugin developer may want to implement a computational task for Kali-ma so that the plugin widget will show a text field for entering the new class name and a combo box for selecting the relationship between classes. The OWL class to associate with the new one with can be captured through the Search widget, while a checkbox may allow the user to decide whether she wishes to have the plugin annotate the new class automatically.

It is also possible for developers to specify the type of the object produced in output by a computational task, if any. With this done, provided that the wrapped function does actually generate such an object, this is presented to the Kali-ma dashboard system, in a similar fashion as items selected through the Search widget. By leveraging this capability, the behaviour exemplified in the paragraph above can be cascaded. Once the output of a plugin is exposed to the system, any other plugin widget that is listening for an object of a compatible type will capture it as an input to its own computational task. This way, it is possible to implicitly define data pipelines in order to accomplish more complex tasks, even if end users or plugin developers are not required to manually concatenate plugin widgets with one another.

Chapter 3

Conclusion

This work has illustrated the key advancements proposed for the Kali-ma plugin a few months since its initial release, which actually occurred with several of these new functionalities actually being in incubation. After validating the interaction approach adopted for the tool, we proceeded to consolidate its user interface and integration with the NeOn Toolkit, offering alternative views on plugin organization, additional support for accessible plugin functionalities and integration with the NeOn Toolkit metalevel, which in turn encompasses its OWL metamodel.

Most significantly, the update introduces a twofold aspect of collaboration, i.e. between users with the introduction of realtime chat support, and between tools with the disclosure of an API for limited control over the Kali-ma dashboard and the extension of plugins for interoperability within the system. The extensibility features were designed so as to require developers not to tamper with their already implemented functionalities, but rather to wrap them in order to make them available to an entry point for Kali-ma to exploit.

With the features described here already in place, our strategy for near future development will mainly focus on tuning up the functionalities proposed both in the first release and in this release. Rather than including whole new, full-fledged functionalities, we aim at evolving existing ones in a manner that will have Kali-ma span across a larger set of potential user demands (in more informal parlance, to make widgets more powerful without necessarily adding new ones). Our goal is to emerge with an application with ample support for the collaborative management of data and metadata of an ontology development project, including their annotation, search, reuse and argumentation. Moreover, enhancing the Semantic Web power of the tool, beyond an initial classification of plugins, is another key task. In particular, we plan on lifting type support for computational tasks and pipelining, from the current level of the Java language to a comprehensive mapping with an ontology network aligned with C-ODO Light. With that done, the system may automatically provide adequate Java classes in line with the input and output knowledge types declared in the OWL descriptions of plugins. Finally, special attention will be devoted to tracking the advancements of the Wave protocol, API and application support in order to establish if they live up to promises and expectations (public disclosure of the federated communication protocol, improved browser compatibility, etc.) in reasonable time. This evaluation will, if negative, be parallel to the setup of a fallback strategy that will most likely include the development and/or deployment of our own communication services.

As a follow-up to the questionnaire-based evaluation held for the first version of the tool, a second, task-based usability study is planned. User interface efficiency will be measured against metrics that will be established with respect to certain tasks, such as performing methodological activities or switching across GUI environments pertaining to certain plugins. A comparison of repeated tasks, either by directly accessing related tools or with the mediation of Kali-ma, will be performed with regard to timings and the overall quality of interaction.

Appendix A

Usage of the Kali-ma interoperability API

This section illustrates details and guidelines for plugin developers who wish to extend their tools in order to be supported by Kali-ma for interoperability purposes. The two types of API are completely independent of each other, and there is no obligation to utilize both or neither. Developers are free to choose to implement either method individually, and the two are not intended to interoperate with each other in general, though developers are free do decide to do so if needed.

As with most Java APIs, as well as with the Eclipse extension point mechanism, the use of types defined in the interoperability API implies a dependency on the Kali-ma plugin that contains their specifications. While it is comprehensible that third parties may not wish to establish a direct dependency of their code on another plugin, this process is essentially inevitable. As a possible workaround to this potential drawback, we advise developers to take advantage of Eclipse *features* in order to create distinct plugin packages.

Figure A.1 depicts a possible strategy for implementing what is being hinted at. Developers are advised to concentrate all of their extensions that depend on Kali-ma into a second plugin (`org.example.myplugin.kalima` in the figure) that bridges the two without necessarily having to tamper with the original (`org.example.myplugin` in the figure). This plugin should depend on the original MyPlugin and the Kali-ma plugin that provides the API that the developer wishes to implement, and it should provide the extension(s) for either dashboard control, or computational tasks, or both. For example, it could contribute to a menu in the original plugin with an additional menu item called "Open as Kali-ma dashboard" (cf. Section A.1), and/or contain the class that implements the extension for the computational task extension point (cf. Section A.2). Then, it would be possible to package the two of them into a Kali-ma-compatible feature, in parallel with an independent feature that does not include the bridging plugin.

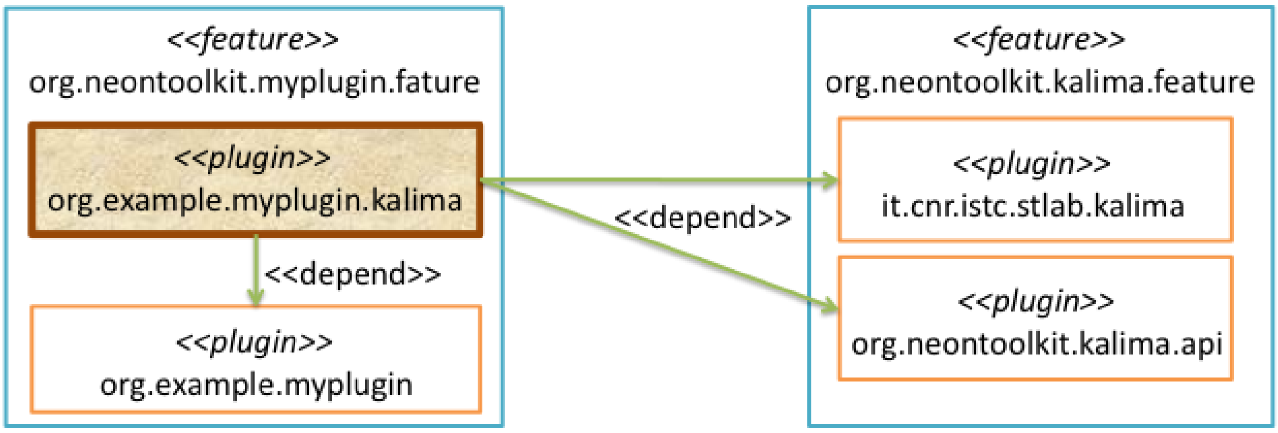


Figure A.1: Suggested layout for plugin dependencies when a generic plugin implements any of the Kali-ma APIs. It is preferable, for the sake of independence, that a bridging plugin (here called `org.example.myplugin.kalima`) be created. Note: Kali-ma no longer depends on external OWL API and Pellet plugins, as it is now fully integrated with the NeOn Toolkit datamodel and its Reasoner plugin.

A.1 Dashboard control

The Dashboard control API allows any other NeOn Toolkit or Eclipse plugin to launch the Kali-ma dashboard and open a pre-defined set of widgets, with options for having them docked from the start and enforcing a particular classification criterion. This set of widgets, along with the name supplied, will make up a temporary dashboard profile, until the user decides to save it as such.

This API is essential and its usage straightforward. To use the API, the plugin developer must:

1. include a dependency on the `it.cnr.istc.stlab.kalima` plugin;
2. invoke any static method of the `it.cnr.istc.stlab.kalima.api.DashboardLauncher` class anywhere in their code.

The most generic method signature in the `DashboardLauncher` class is

```
launchDashboard(String profileName, String[] pluginIDs,
    ClassificationCriterion criterion, boolean docked)
```

where:

- `profileName` is a non-null, non-empty identifier for this set of plugins. It will be proposed as a name for the profile, although users will be free to give it a different name.
- `pluginIDs` is an array of unique identifiers for the plugins to be opened as widgets. The Kali-ma plugin identifier should not be part of this list, and will be ignored otherwise. Redundancies will be counted as a single widget. The array can be null or empty, in which case a blank Kali-ma dashboard will be opened as if the Kali-ma plugin itself were launched the regular way.
- `criterion` is a member of the `ClassificationCriterion` enumeration, available in the same package as the `DashboardLauncher` class. Its value can be one of `DESIGN_ASPECTS`, `FUNCTIONALITIES` or `NEON_ACTIVITIES`. This argument causes the corresponding classification criterion to override user settings when generating the C-ODO organizer. If null, the criterion defined by user preferences will be used instead. If Kali-ma is already running, this change will only take effect after restarting the NeOn Toolkit.
- `docked`, if true, requests all plugin widgets to initially appear minimized as entries in the Dock widget.

For example, a call to the method `DashboardLauncher.launchDashboard("scheduling + reuse", {"org.neontoolkit.upm.gontt", "org.neontoolkit.watsonplugin"}, ClassificationCriterion.NEON_ACTIVITIES, false)` will cause Kali-ma to launch its dashboard by presenting an overview of plugins classified by activity in the NeOn methodology, with two widgets for the gOntt and Watson plugins visible in the dashboard, and the “scheduling + reuse” title displayed on the Profile management widget.

The `criterion` and `docked` arguments are optional, as the `launchDashboard` method is overloaded with additional signatures that allow developers to not specify either argument. An unspecified `criterion` is interpreted as `null`, while an unspecified `docked` is interpreted as `true`. See above for their meaning.

Note that dashboard creation is synchronized with the UI thread, hence developers should not worry about managing that kind of synchronization themselves.

A.2 Computational tasks

Through the Computational task API, developers can provide extensions to functionalities for which code already exists, or wrap part of such functionalities, or even create new ones from scratch, if they believe them to pertain to the use cases specified for their plugins. These functionalities can then be invoked from within the Kali-ma dashboard, and its input parameters will be presented as user interface elements (or *controls*) for users to input where applicable, or as "invisible" listeners when such datatypes are not mappable to user interface elements.

As opposed to the dashboard control API, computational tasks are implemented through the extension of an Eclipse *extension point* specifically provided by Kali-ma. Extension points are the means by which the Eclipse RCP allows developers to add functionalities and UI elements to the platform they are contributing to[1]. Extensions for this extension point are then scanned at runtime when the Kali-ma plugin starts, just like those that implement views, perspectives, actions or wizards.

As with the majority of standard extension points provided by Eclipse, the Kali-ma extension point requires an interface to be implemented, along with its only method. This is slightly more complex than a straightforward call to an API method, though seasoned plugin developers will be familiar with it. However, since developers are most likely to wrap up existing functionalities, not much work should be required on their part for achieving this goal.

To implement a computational task, the following steps are required for the implementing plugin:

1. include a dependency on the `org.neontoolkit.kalima.api` plugin;
2. add an extension to the `it.cnr.istc.stlab.computationalTasks` extension point (can be done either through the Extensions GUI for that plugin or by manually editing the `plugin.xml` manifest file). Setting an ID and/or a name for the extension is preferred but not required;
3. add a `task` configuration element to the newly generated extension, and create the (required) Java class that implements the `IComputationalTask` interface;
4. if the extension is being implemented in an external plugin, add the main plugin ID to the `referenced_plugin` attribute of the `task` configuration element created at the previous point;
5. implement the `execute()` method in the class created at point 3 (see later for details), and annotate it with the `Signature` annotation (available from the newly imported package).

The last item in the checklist requires a more thorough explanation.

The `IComputationalTask` interface has a single `execute()` method, which is automatically invoked by Kali-ma. It has the following modifiers and signature:

```
public Object execute(Object...  params)
      throws UnexpectedParameterTypeException
```

This method is highly versatile, as it expects a variable number of generic arguments (even zero) as input and return a generic `Object` (even null) as an output. This is done in order to accommodate virtually any kind of procedure and allow developers to use this method to emulate a call to that procedure.

Obviously, these generic `Object`s will have to match a limited amount of more specific types. It is fine (and encouraged) to perform type checking in the method body, but also Kali-ma has to be made aware of these types in order to present the appropriate UI controls. Therefore, developers are required to annotate their implementation of the `execute()` method with the `Signature` annotation, which allows them to specify the actual types of both the input arguments (as an array of `java.lang.Class` objects) and the

[1]For a tutorial on Eclipse extensions and extension points, see http://www.vogella.de/articles/EclipseExtensionPoint/article.html

return value (as a single `java.lang.Class` object). Therefore, when implementing a computational task, developers should keep the following principles in mind:

1. Always annotate an implemented `execute()` method with a truthful `Signature`.
2. If the procedure does not return anything, the annotation's `returnType` parameter should be assigned to `void.class`. The `execute()` method should return something anyway, even `null`. Although any return value will be ignored in this case, this is necessary to avoid compile-time errors.
3. The `execute()` method should basically call a public method that performs the desired functionality, and convert its input and return types accordingly with the `Signature` annotation. If there is no such public method, then one should be made available. Alternatively, the `IComputationalTask` interface could be implemented by the very same class that does the job. In this case, however, it will no longer be possible to implement extensions or Kali-ma as a separate plugin.
4. The method should check that every parameter supplied is of the required type, and throw an `UnexpectedParameterException` every time it doesn't.

Currently, only a few basic and OWL API-related types are managed by Kali-ma:

- `String` objects are mapped to *text fields*;
- `Integer` or even `int` objects are mapped to *spinners*;
- `Boolean` or even `boolean` objects are mapped to *checkboxes*;
- `enum`s are mapped to *combo boxes*. Any enumeration can be arbitrarily defined by developers in the same class that implements the `IComputationalTask` interface;
- `File` objects are mapped to *file selectors*;
- instances of OWL API types such as `OWLEntity` and its subtypes, e.g. `OWLClass`, `OWLNamedIndividual`, `OWLObjectProperty`, etc. are not mapped to any visible interface control, but the widget will be automatically put on a listening state in order to get a handle on any such entity that is presented to the system through another widget;
- Arrays of the types mentioned above are allowed as return types, and are mapped to expansible lists.

For clarity, the Java code listing of a possible computational task implementation for the Watson plugin is included below. Refer to Javadoc and inline comments for an explanation of each step of the procedure and how the `Signature` annotation will be interpreted by Kali-ma.

```
package uk.ac.open.kmi.watson.neontoolkitplugin.procedures;

import it.cnr.istc.stlab.kalima.api.IComputationalTask;
import it.cnr.istc.stlab.kalima.api.Signature;
import it.cnr.istc.stlab.kalima.api.UnexpectedParameterTypeException;

import org.semanticweb.owl.model.OWLEntity;

import uk.ac.open.kmi.watson.clientapi.WatsonService;
import uk.ac.open.kmi.watson.neontoolkitplugin.actions.WatsonControl;
import uk.ac.open.kmi.watson.neontoolkitplugin.utils.LabelSplitter;

public class WatsonQueryTask implements IComputationalTask {

    /**
     * This enumeration was created specifically for having a choice of three
     * possible values in the combo box.
     *
     */
    public enum EntityType {
        Class, Property, Individual
    };

    public WatsonQueryTask() {
        // A constructor that needs to be called by Kali−ma
    }

    /**
     * As stated in the Signature annotation, Watson consumes an OWL entity and
     * a choice of Class, Property and Individual, and produces OWL entities
     *
     * Because it only expects an OWLEntity as an input, the widget will show a
     * combo box interface control for the second argument, while for the first
     * it will listen for OWL entities to be presented in the Kali−ma system.
     */
    @Signature(
            parameterTypes = { OWLEntity.class, EntityType.class },
            returnType = OWLEntity[].class
    )
    public Object execute(Object... params)
            throws UnexpectedParameterTypeException {

        // parse the first two arguments (the only ones needed)
        Object arg0 = params[0], arg1 = params[1];

        String[] keywords;
        int entityType = −1;

        // do some type checking on the argument(s)...
        if (!(arg0 instanceof OWLEntity && arg1 instanceof EntityType))
            throw new UnexpectedParameterTypeException();

        // Now convert the first argument to an array of strings, which is the
        // form required by the internal method to be called.
        // This bit was copied straight from existing Watson plugin code.
        //
        // extract the keyword array from the entity name...
        String baseKW = ((OWLEntity) arg0).getURI().getFragment();
        // split it to an array
        String st = new LabelSplitter().splitLabel(baseKW);
        // remove unwanted characters
        if (st.endsWith("/"))
            st = st.substring(0, st.length() − 1);
        st = st.replaceAll("/", "−");
        st = st.replaceAll("'", "");
        keywords = new String[] { st };

        // Now do the same with the second argument
        switch ((EntityType) arg1) {
        case Class:
            entityType = WatsonService.CLASS;
            break;
        case Individual:
            entityType = WatsonService.INDIVIDUAL;
            break;
        case Property:
            entityType = WatsonService.PROPERTY;
            break;
        }

        // If the types are consistent, make your internal calls
        //
        // NOTE: I added the queryWatsonAsEntities() method to the WatsonControl
        // class. It just returns an array of OWLEntities out of a Watson query.
        // Nothing out of the ordinary, really. Just a wrapper method for
        // WatsonControl.queryWatson(). Alternatively, the IComputationalTask
        // interface could have been implemented straight into the WatsonControl
        // class.
        return WatsonControl.getInstance().queryWatsonAsEntities(keywords,
                entityType);
    }

}
```

Bibliography

[AGP09] Alessandro Adamou, Aldo Gangemi, and Valentina Presutti. C-ODO plugin v1.0. Deliverable D2.3.4, NeOn project, 2009.

[BOCGP04] Jesús Barrasa, Óscar Corcho, and Asunción Gómez-Pérez. R2O, an extensible and semantically based database-to-ontology mapping language. In *Proceedings of the 2nd Workshop on Semantic Web and Databases(SWDB2004*, pages 1069–1070. Springer, 2004.

[CL08] Germán Herrero Carcel and Tomás Pariente Lobo. Revision of ontologies for Semantic Nomenclature: pharmaceutical networked ontologies. Deliverable D8.3.2, NeOn project, 2008.

[DEB+08] Klaas Dellschaft, Hendrik Engelbrecht, José Monte Barreto, Sascha Rutenbeck, and Steffen Staab. Cicero: Tracking design rationale in collaborative ontology engineering. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 782–786. Springer, 2008.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications, 2004.

[GPSFV09] Asunción Gómez-Pérez, Mari Carmen Suárez-Figueroa, and Martin Vigo. gOntt: a tool for scheduling ontology development projects. In *8th International Semantic Web Conference (ISWC2009)*, October 2009.

[PMP+09] Valentina Presutti, Dunja Mladenic, Raul Palma, Klaas Dellschaft, Alessandro Adamou, Enrico Daga, Holger Lewen, Michael Erdmann, and Anne Becker. Practical methods to support collaborative ontology design. Deliverable D2.3.2, NeOn project, 2009.

[Ree79] Trygve Reenskaug. Models - Views - Controllers. Technical report, Technical Note, Xerox Parc, 1979.

[SFBd+09] Mari Carmen Suárez-Figueroa, Eva Blomqvist, Mathieu d'Aquin, Mauricio Espinoza, Asunción Gómez-Pérez, Holger Lewen, Igor Mozetic, Raul Palma, Maria Poveda, Margherita Sini, Boris Villazón-Terrazas, Fouad Zablith, and Martin Dzbor. Revision and extension of the NeOn Methodology for building contextualized ontology networks. Deliverable D5.4.2, NeOn project, 2009.

[SHNM04] Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/JFace in Action: GUI Design with Eclipse 3.0 (In Action series)*. Manning Publications, 2004.

[TNTM08] Tania Tudorache, Natalya Fridman Noy, Samson W. Tu, and Mark A. Musen. Supporting collaborative ontology development in Protégé. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2008.