UNIVERSITà DEGLI STUDI DI FERRARA

FACOLTà DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

PROGETTAZIONE E SVILUPPO DI UN AMBIENTE INTEGRATO DI PROGRAMMAZIONE PER IL LINGUAGGIO SCIFF

Tesi di Laurea di

CANTELMO CHRISTIAN

Relatore:

Prof. Ing. MARCO GAVANELLI

Indice generale

1 Obiettivi

E' risaputo che le fondamenta per la progettazione di un buon software sono basate su criteri tra i quali spiccano i concetti di “estensibilità” e di “usabilità”. Il primo concetto è quanto mai importante e identifica la necessità di poter estendere il prodotto, una volta completato, in modo facile e senza modifiche sostanziali di quanto già sviluppato. Il secondo è un concetto fondamentale e che richiede una particolare attenzione verso l'utente finale e gli schemi già collaudati di certi tipi di interfacce di uso comune.

In questo caso, in particolare, ci si è dovuti confrontare con la progettazione e la realizzazione di una interfaccia grafica di un ambiente di sviluppo basato su un linguaggio in evoluzione: SCIFF [1]. Se è vera in generale la necessità di porre un occhio di riguardo verso i due aspetti accennati all'inizio, in questo caso particolare lo è ancora di più. Si avrà infatti a che fare con la progettazione di un componente che farà da tramite per un linguaggio (SCIFF) che sarà probabilmente futuro oggetto di estensioni; si pone quindi l'accento sulla necessità del componente in questione di essere agevolmente ampliabile e modificabile a sua volta per meglio rispondere alle future esigenze.

Proprio a causa di ciò, una delle scelte critiche nell'approccio allo sviluppo di questa applicazione è stata se sviluppare un prodotto stand-alone, oppure un plug-in da agganciare ad una applicazione preesistente. La scelta infine è ricaduta su quest'ultima opzione e, in particolare, Eclipse [2] è stata scelta come l'applicazione per la quale sviluppare il plug-in in questione.

Le motivazioni che hanno spinto a questo tipo di scelta sono state fondamentalmente due, ed entrambe volte a dare una risposta alle esigenze riportate all'inizio.

La prima, dettata dalla necessità di rendere il plug-in facilmente estendibile, trova riscontro nel meccanismo di estensione fornito dall'ambiente Eclipse. Quest'ultimo, infatti, mette a disposizione dello sviluppatore una serie di punti

di estensione (extension point), i quali, una volta dichiarati nella propria applicazione, ci permetteranno di integrarci al software principale utilizzando le interfacce forniteci. In particolare, essendo ogni punto di estensione relativo ad un aspetto ben specifico dell'applicazione, questa metodologia ci permette ancora più facilmente di affrontare lo sviluppo dei singoli aspetti in modo separato.

La seconda motivazione, dettata questa volta dalla necessità di fornire un'interfaccia facilmente fruibile al nostro plug-in, è soddisfatta dal concetto che sta dietro all'intero progetto di Eclipse, ovvero di essere un ambiente incentrato sullo sviluppo di ambienti integrati di sviluppo (IDE). Questo ci permette, infatti, di poter fruire di molte funzionalità atte a risolvere problematiche ricorrenti in questo ambito di sviluppo e di poterlo fare seguendo certi standard ai quali l'utente è ormai abituato.

Parlando invece di quelli che sono stati gli scopi ultimi di questo progetto, invece, primo fra tutti menzionerei la necessità di facilitare all'utente finale le operazioni più comuni di editing, implementando meccanismi di evidenziazione della sintassi, pulsanti per l'inserimento automatico degli operatori di uso più comune, evidenziazione del bilanciamento delle parentesi e altri strumenti.

In ultima battuta vorrei evidenziare la problematica legata alla necessità di gestire una componente software esterna come SICStus Prolog [3](su cui SCIFF è basato) ed integrarlo all'IDE nel modo più trasparente possibile garantendone però un funzionamento sincronizzato alle altre componenti.

2 Background

2.1 SCIFF

E' utile, innanzitutto, una breve introduzione sul linguaggio SCIFF, al quale verranno dedicate le funzionalità di questo plugin.

Il linguaggio SCIFF si basa su una procedura di dimostrazione abduttiva, la quale permette di formulare delle ipotesi per descrivere le osservazioni effettuate. SCIFF, infatti, originariamente era nato come una procedura atta a verificare che agenti facenti parte di una società aperta, rispettassero dei vincoli dettati da un protocollo specificato secondo una determinata sintassi.

I file necessari per l'utilizzo della procedura sono di tre tipi:

- i file History che rappresentano la serie di eventi osservati nella società di cui si osserva il comportamento
- i file IC's che rappresentano i vincoli da seguire per il rispetto del protocollo
- i file sokb nei quali è contenuta la base di conoscenza della società in esame

Tutti i file che rappresentano l'applicazione saranno raggruppati in un unico progetto. Di questo verranno scritte tutte le informazioni all'interno di un file denominato project.pl.

Inoltre, SCIFF prevede l'impostazione di una serie di opzioni per ogni singolo progetto, le quali ne modificheranno il comportamento in particolari ambiti o estensioni.

2.2 Eclipse

Quando si parla di Eclipse[4] spesso ci riferiamo ad Eclipse Software Development Kit (SDK) il quale, in realtà, non è altro che una serie di tool basati sulla vera e propria Eclipse Platform (Fig. 1). Questi tool comprendono

ad esempio il Java Development Tool (JDT), un ambiente di sviluppo per il linguaggio Java, ed il Plug-in Development Environment (PDE), un ambiente di sviluppo di plug-in per lo stesso Eclipse, oltre che la già citata Eclipse Platform.

E' importante sottolineare come la Eclipse Platform sia a sua volta un insieme di componenti progettati per fornire tutti gli strumenti necessari alla creazione di ambienti di sviluppo facilmente estensibili e integrati. Un sottoinsieme di questi, detto Eclipse Rich Client Platform (RCP), ci fornisce gli strumenti di sviluppo per costruire applicazioni stand-alone di qualsiasi tipo, allargando notevolmente l'ambito di utilizzo della piattaforma.

In particolare, uno dei punti di forza di Eclipse è la concezione della Platform come punto di integrazione. Costruendo un applicazione sulla Platform, infatti, la si integra con altre applicazioni a loro volta sviluppate con lo stesso criterio.

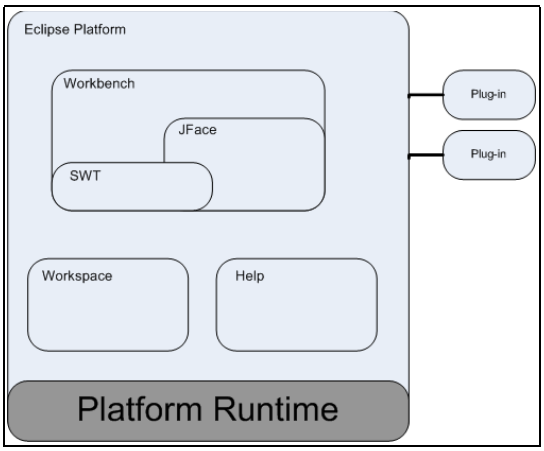

Fig. 1: Eclipse Platform

2.2.1 Plug-in

Eccezione fatta per la Platform Runtime, tutte le funzionalità di cui possiamo usufruire in Eclipse sono parte di un Plug-in. Questa architettura può,con l'esigenza di funzionalità aggiuntive e le necessità crescenti, portarci ad avere

anche considerevoli quantità di moduli installati sulla nostra piattaforma. Per ovviare ad un suo appesantimento, è stato sviluppato un meccanismo che permette di caricare solamente il codice dei Plug-in di cui l'utente fa effettivo utilizzo.

Più nello specifico, ogni Plug-in è dotato di un file manifesto il quale specifica i legami del primo con altri Plug-in in termini di estensioni di funzionalità (extensions), fornendo con esse informazioni aggiuntive scelte dallo sviluppatore dell'estensione. Ogni Plug-in, a sua volta, può implementare dei propri punti di estensione (extension-point) ai quali altri moduli si agganceranno per espanderne le funzionalità.

Allo start-up della piattaforma, il modulo Platform Runtime si occupa di verificare la lista dei Plug-in disponibili, leggere i loro file-manifesto e crearne un registro. In particolare, quello che preserva la leggerezza della piattaforma è il fatto di poter leggere le informazioni riguardanti le funzionalità implementate da un certo plug-in e dichiarate sul proprio file-manifesto, accedendo esclusivamente a quest'ultimo, senza essere costretti a caricare il codice relativo all'applicazione in questione.

Il codice del plug-in verrà caricato solamente quando ve ne sarà la necessità, ovvero, non verrà caricato fino a quando l'utente non farà effettivo utilizzo di una delle sue funzionalità. Avverrà così l'effettivo caricamento del plug-in, che, una volta attivato, rimarrà in memoria fino alla chiusura della Platform o alla sua esplicita disattivazione.

2.2.2 Workspace

Il workspace rappresenta lo spazio di lavoro sul quale è possibile agire attraverso gli strumenti forniti dalla piattaforma. Questo consiste in una serie di progetti, ognuno dei quali è collegato ad una directory specifica nel file system. Di default tutti i progetti sono contenuti in una unica directory di workspace.

La piattaforma fornisce anche la possibilità di caratterizzare ogni progetto con

un attributo “nature” il quale permetterà ad Eclipse di distinguerlo dagli altri, ed, eventualmente, attribuirgli un comportamento specifico. Ogni progetto potrà avere un numero di “nature” che varia a seconda delle esigenze.

All'interno di ogni progetto saranno presenti file e folder i quali saranno accessibili anche dagli altri programmi installati sul sistema operativo sottostante la piattaforma in modo completamente trasparente.

Per prevenire eventuali rischi legati alla perdita di dati esiste un meccanismo di “history” che tiene traccia delle modifiche apportate dai tool interni alla piattaforma ai file e dei loro precedenti contenuti.

Un'altra caratteristica interessante consiste nella possibilità di apporre dei “marker” alle risorse del workspace di cui ci si può servire per particolari annotazioni, quali, ad esempio, la segnalazione di un errore di sintassi in una data riga di codice.

In aggiunta viene fornito anche un meccanismo più generale che permette al plug-in di tenere traccia di eventuali modifiche al workspace registrando un apposito “resource change listener”, attraverso il quale è possibile ricevere la notifica delle modifiche apportate alle risorse non appena avvengono. Ogni notifica ci verrà fornita sotto forma di “delta tree”, ovvero un albero con root nella “workspace root” della piattaforma (ovvero la cartella contenente tutti i progetti del workspace), all'interno del quale saranno presenti i file che avranno subito modifiche e le directory che li contengono. Le modifiche notificate comprendenti cancellazione, creazione, modifica di markers e del contenuto dei files, saranno rese visibili all'interno del “delta tree”.

2.2.3 SWT

Lo Standard Widget Toolkit (SWT) fornisce una serie di interfacce indipendenti dal sistema operativo per l'implementazione di widgets (button, checkbox, label...) ed elementi grafici. Rispetto alle librerie Java AWT e SWING fornisce una maggiore integrazione con il sottostante window system mantenendo al tempo stesso una grande portabilità.

2.2.4 Jface

Jface è un toolkit sviluppato interamente in Java che si occupa di fornire allo sviluppatore molti elementi grafici di uso comune di alto livello attraverso classi già pronte. Gli elementi forniti allo sviluppatore sono molteplici: action, view wizard e altri.

In particolare:

- Action: rappresenta un comando che può essere richiamato tramite un elemento dell'interfaccia grafica. Anche in questo caso abbiamo una separazione tra l'ambito della presentazione e quello comportamentale. Avremo, infatti, che le informazioni necessarie alla rappresentazione grafica dell'elemento saranno presenti nel file manifesto del plug-in, in modo da rendere sempre visibile il componente grafico all'utente, mentre la classe che implementa il lato comportamentale sarà richiamata solo una volta che si sarà attivato il determinato comando. Questa separazione ci permette di modificare comodamente uno dei due aspetti senza compromettere il funzionamento dell'altro.
- View: sono elementi grafici basati sul design pattern Model-Viewer-Controller (MVC). Questo pattern definisce la cooperazione fra tre tipologie di componente: il componente Model gestisce il dominio dei dati, Viewer è responsabile della rappresentazione dei dati e il componente Controller si occupa della gestione dell'interazione con l'utente. Questa architettura diverrà molto comoda se, ad esempio, si vorrà decidere di fornire diversi tipi di rappresentazione per lo stesso tipo di dato.

2.2.5 Workbench

Al contrario delle librerie SWT e Jface di uso generico, il workbench fornisce

al plug-in l'aspetto e la personalità della piattaforma Eclipse. E' interessante sottolineare come, però, il Workbench sia stato implementato facendo uso esclusivamente delle prime, escludendo l'utilizzo delle librerie Java AWT e SWING.

La struttura dell'interfaccia grafica di Eclipse si basa su tre elementi principali: editor, view e perspective.

Gli editor ci permettono di aprire, modificare e salvare file, le view ci permettono di visualizzare il contenuto di un oggetto attualmente in uso in modo strutturato o di visualizzare qualsiasi altro dato ad esso relativo, mentre le perspective non sono altro che una selezione delle prime due tipologie di elementi che l'utente vedrà visualizzata all'interno dell'interfaccia. All'interno del Workbench possono coesistere più perspective in una volta sola, una sola di esse, però, potrà essere visualizzata in un dato istante.

3 Manuale di installazione e utilizzo

3.1 Installazione

Procediamo col mettere in evidenza i requisiti necessari per poter utilizzare al meglio il plug-in.

Occorre innanzitutto avere installato SICStus Prolog versione 3.10.1 o successiva, il quale sarà il motore su cui poggerà la fase di esecuzione dei progetti .

Il passo successivo sarà quello di operare il download dei file del linguaggio SCIFF, reperibili presso il sito internet ufficiale all'indirizzo http://lia.deis.unibo.it/research/sciff/ utilizzando l'apposito link, e decomprimere la cartella che li contiene su disco.

La fase successiva, prevede l'installazione dell'ambiente Eclipse, scaricabile gratuitamente presso il sito internet ufficiale http://www.eclipse.org/ nella sezione “download”.

Una volta installati SICStus Prolog, SCIFF ed Eclipse, si potrà procedere all'installazione del plug-in. Questa, consisterà nel copiare la directory denominata “SCIFF_1.0.0”, contenente il plug-in, nella sottodirectory “plugins” della cartella di installazione di Eclipse. A ciò, dovrà seguire,esclusivamente per la prima esecuzione dopo l'installazione del plug-in, l'avvio di Eclipse in modalità clean, tramite il comando “eclipse -clean”.

Una volta avviato Eclipse, ci verrà chiesto di specificare la directory in cui salvare i nostri progetti. Il percorso di quest'ultima dovrà essere specificato anche nel file `defaults.pl` nella directory di SCIFF, e, per la precisione, all'interno del predicato `default_dir`.

Fatto questo, potremo iniziare ad utilizzare il plug-in.

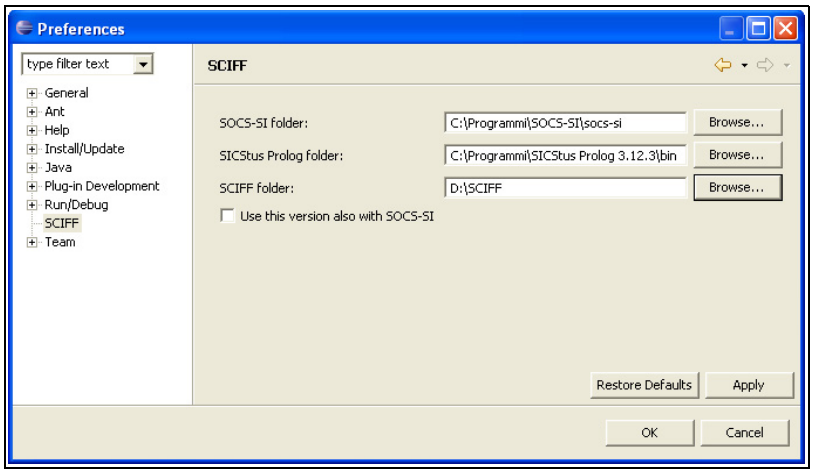

Fig. 2: Preferenze

3.1.1 Settaggio delle prime impostazioni

Occorre, una volta avviato Eclipse, settare alcune impostazioni necessarie a permettere un corretto utilizzo di tutte le funzionalità del programma. Per fare questo, è necessario comunicare al plugin le directory in cui sono installati i programmi che devono essere invocati, cioè SICStus Prolog, SCIFF e SOCS-SI [5].

Per fare questo, si utilizza il menù Window -> Preferences -> SCIFF (Fig. 2).

In particolare, il completamento del campo “SOCS-SI folder” sarà necessario solo nel caso si abbia installato nel file system SOCS-SI e si abbia intenzione di integrarlo in Eclipse.

3.2 Utilizzo

3.2.1 Creare e importare un progetto

Il primo passo per iniziare a lavorare all'interno di Eclipse con un progetto SCIFF, sarà quello di creare un nuovo progetto all'interno dell'Eclipse workspace oppure di importarne uno già esistente. Per entrambe le operazioni sono presenti due wizard che ne faciliteranno lo svolgimento.

Per quanto riguarda la creazione di un nuovo progetto, si avranno due possibilità per accedere al wizard apposito: la prima consiste nella selezione del menù “File”->”New”->”Other”->“Project”, l'alternativa invece, disponibile solo nel caso sia già attiva la “SCIFF perspective” (di cui si parlerà nel

prossimo paragrafo), prevede di cliccare con il tasto destro del mouse in corrispondenza della finestra di navigazione sulla sinistra del workbench e selezionare la voce "New"->"Project".

L'operazione appena svolta permetterà la visualizzazione di una finestra dove verrà richiesto l'inserimento del nome del progetto che si intende creare all'interno del campo testuale "Project name". Una volta inserito il dato, premendo il tasto "Finish" verrà avviato il processo di creazione del nuovo progetto.

All'interno di ogni nuovo progetto, viene creato di default:

- un file progetto "project.pl"
- tre directory: "ICS", "HISTORY" e "SOKB"

Per importare un progetto di SCIFF già esistente, invece, occorrerà selezionare dal menù "File" la voce "Import..." -> "SCIFF project" e inserire il percorso del progetto che si intende importare.

L'operazione di importazione di un progetto già esistente effettua la scansione di tutti i file presenti nella directory di origine e, a seconda che nel loro nome sia presente la stringa "ics","sokb" o "history", li assegna alla corrispondente tipologia di file, combiando l'estensione rispettivamente in ".ics", ".sokb" e ".his". Fatto questo, i file verranno inseriti in una cartella corrispondente al tipo di file fra le tre che vengono create di default, ovvero: "ICS", "HISTORY" e "SOKB".

E' importante sottolineare come il file project.pl del progetto importato verrà modificato solo aggiornando il path dei file riportati al suo interno.

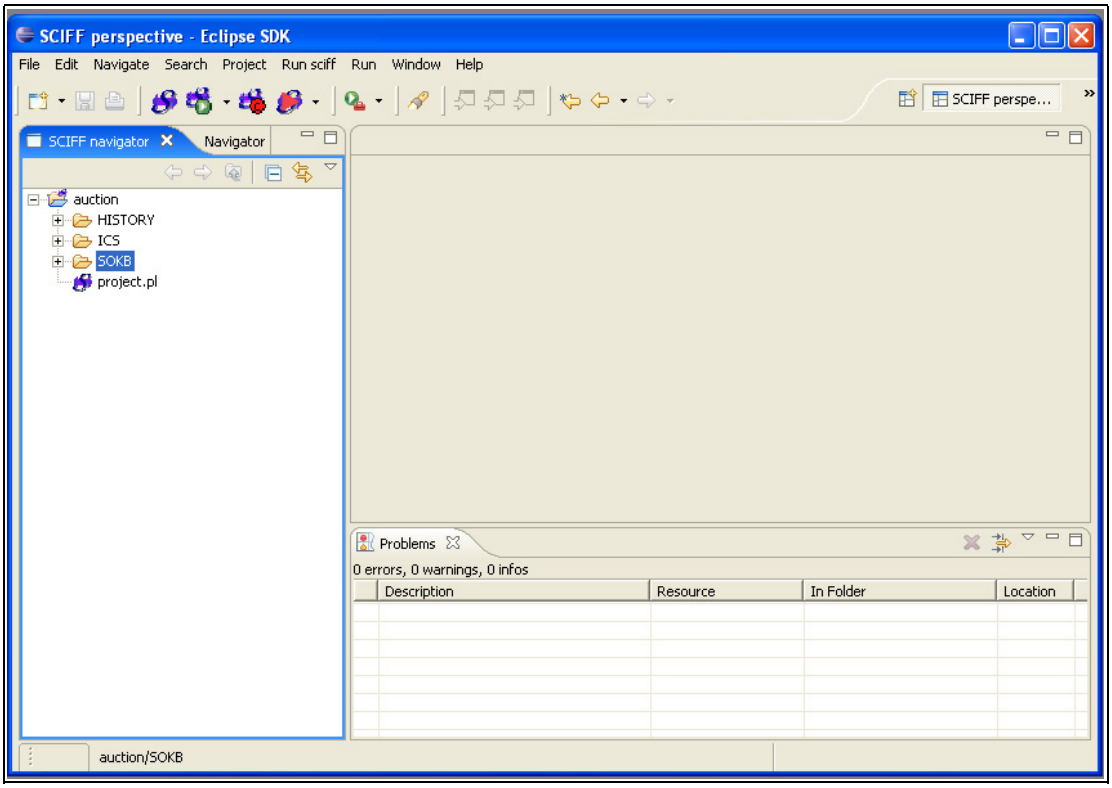

Fig. 3 : SCIFF perspective

3.2.2 SCIFF perspective

Una volta portata a termine la procedura per l'importazione o la creazione di un nuovo progetto, verrà immediatamente aperta una nuova perspective (Fig. 3), denominata "SCIFF perspective", che ospiterà le finestre di lavoro per quanto concerne i progetti SCIFF.

All'interno di quest'ultima, inizialmente, si potranno individuare tre zone adibite alla visualizzazione di varie finestre: una, sulla sinistra, contenente le view "SCIFF navigator" e "Navigator", una nella parte inferiore contenente la view "Problems" e, infine, una centrale inizialmente vuota, ma adibita ad ospitare gli editor.

La "SCIFF navigator" consentirà di visualizzare la struttura ad albero dei soli progetti SCIFF attualmente presenti nel workspace. La Navigator view visualizzerà, invece, tutti i progetti presenti nel workspace, indipendentemente dalla loro tipologia.

La problems view, infine, consentirà la visualizzazione dei problemi riscontrati in fase di lancio del progetto SCIFF. Essa permetterà anche, tramite doppio click sulla riga di riferimento del problema, un rimando alla zona di codice

interessata.

3.2.3 Importare file nei progetti

Per importare qualsiasi tipo di file all'interno di un progetto di SCIFF, si potrà utilizzare il wizard apposito. Per accedervi, occorrerà selezionare dal menù "File" la voce "Import...". Questa permetterà di accedere ad una finestra di wizard che richiederà la scelta del tipo di risorsa che si intende importare. Dopo aver selezionato la voce "SCIFF file" e avere premuto il tasto "Next", troveremo tre campi diversi:

- nel primo si dovrà indicare quale sarà il progetto di destinazione
- nel secondo il file di origine, impostabile sia digitandolo nel campo di testo, sia selezionandolo tramite la finestra di navigazione del file system.
- il terzo campo ci permetterà di decidere se importare il file come file "Generico", ovvero mantenendone l'estensione originale, oppure come "Ics", "Sokb", ".ruleml" o "History". Negli ultimi casi, l'estensione del file di origine verrà mutata in quella predefinita per uno di questi tipi di file.

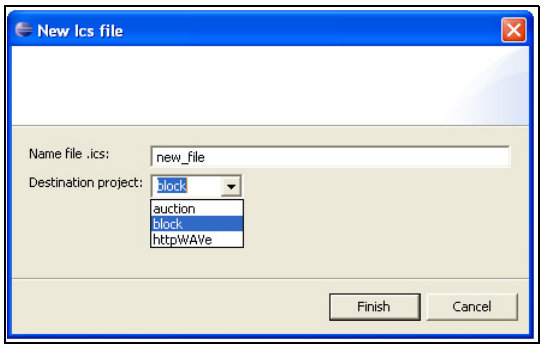

Fig. 4: New ICS wizard

3.2.4 Creare nuovi ics, sokb e history file

Per inserire un nuovo file nel progetto sul quale si sta lavorando, si potrà utilizzare l'apposito wizard (Fig. 4), per accedere al quale sono possibili due strade:

- dopo aver aperto il menù "File", selezionare la voce "New" ed in

seguito “Sokb file”,”Ics file” o “History file” a seconda del tipo di file che si intende creare.

- Cliccare col tasto destro del mouse sulla finestra “SCIFF navigator”, selezionare la voce “New” ed in seguito “Sokb file”, ”Ics file” o “History file” a seconda del tipo di file che si intende creare.

A questo punto verrà richiesto il nome del file da creare e all'interno di quale progetto il file dovrà essere creato.

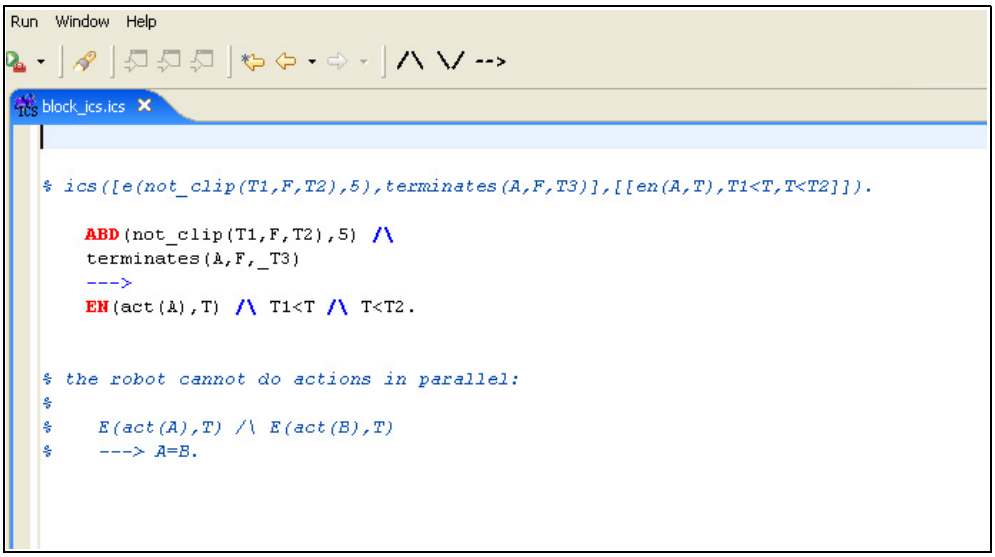

Fig. 5 : Ics editor

3.2.5 Ics, Sokb e History editor

Il linguaggio SCIFF prevede l'utilizzo di tre tipologie di file che presentano una sintassi differente. Questo è gestito dal plug-in tramite tre editor differenti, dei quali uno è illustrato in Fig. , permettendo così una più efficace gestione della rappresentazione del codice.

Gli aiuti visivi che i vari editor mettono a disposizione dell'utente sono i seguenti:

- l'evidenziazione del bilanciamento delle parentesi, ovvero, nel momento in cui il cursore si troverà nella posizione successiva ad una parentesi, la sua relativa (di chiusura o di apertura) sarà evidenziata da una cornice.
- l'evidenziazione della sintassi

- suggerimenti sugli argomenti da inserire in un dato termine all'interno di riquadri che appaiono dopo l'apertura della parentesi del termine dato e scompaiono dopo la chiusura della sua relativa.

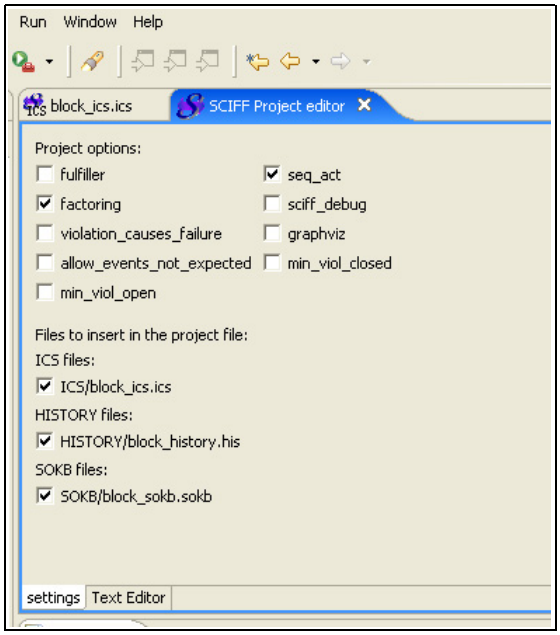

Fig. 6 : project editor

3.2.6 Project.pl editor

Per quanto riguarda il file di progetto di SCIFF, identificato dal file "project.pl", è possibile modificarlo utilizzando, nello specifico, l'editor "SCIFF project editor" (Fig. 6).

Questo apparirà come una finestra con due pagine accedibili separatamente tramite due linguette apposte ai piedi della finestra.

In particolare la prima, denominata "settings", permetterà un più veloce editing sulle opzioni selezionabili per il progetto e i file da includere in quest'ultimo nella sessione di lancio. Tutto questo sarà possibile farlo tramite la selezione di una serie di checkbox.

La seconda, denominata "Text editor", sarà un vero e proprio editor di testo che ci consentirà di modificare più direttamente, e quindi più in dettaglio, il file.

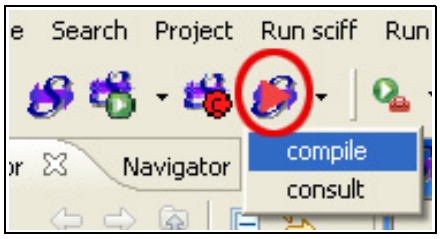

Fig. 7: start SCIFF

3.2.7 Run SCIFF

Una volta conclusa la fase di editing, si vorrà procedere al lancio del programma appena scritto. Per fare questo, si dovrà, prima di tutto, attivare l'interprete SCIFF, il quale sarà avviabile specificando a SICStus la direttiva "consult" o "compile".

Questa scelta si potrà effettuare tramite il menù a scomparsa attivabile premendo il pulsante illustrato in figura 7.

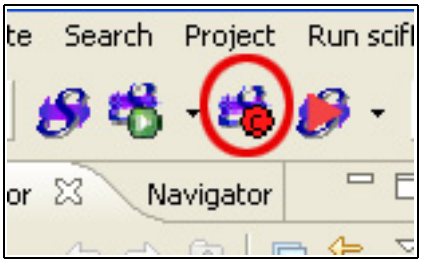

Fig. 8: compile SCIFF project

3.2.8 compilazione di un progetto SCIFF

Dopo aver avviato SCIFF, nel caso in cui si volesse procedere alla compilazione di un progetto presente nel workspace, basterà premere l'apposito pulsante illustrato in figura 8.

Una volta premuto, comparirà una finestra denominata "run SCIFF project", la quale permetterà di selezionare il progetto che si intende lanciare. La scelta avverrà solamente su un elenco di progetti SCIFF, gli altri presenti nel workspace, infatti, non saranno elencati.

A selezione avvenuta, si potrà procedere alla compilazione del progetto premendo il pulsante OK della finestra.

Nel caso di errori durante la compilazione del progetto, il processo in corso verrà terminato e nella finestra "Problems" ne verrà visualizzata la causa.

In particolare, facendo doppio click sulla riga relativa all'errore, potremo visualizzare sull'editor corrispondente la sezione di codice che lo ha causato.

Fig. 9: run SCIFF project

3.2.9 Lancio di un progetto SCIFF

Effettuata anche la compilazione del progetto, potremo procedere al suo lancio nelle tre modalità previste da SCIFF. Queste saranno accessibili tramite il menù a scomparsa attivabile premendo il pulsante illustrato in figura 9.

A default, `run` equivale a `run_closed`; tuttavia può essere ridefinito dall'utente nel file project.pl per ottenere comportamenti diversi da quelli standard.

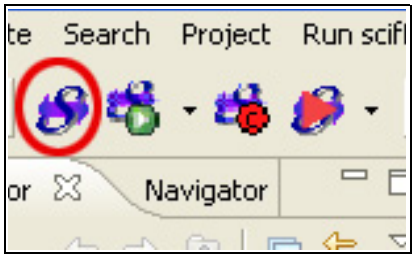

Fig. 10: run SOCS-SI

3.2.10 Run SOCS-SI

Come accennato in precedenza, nel caso si possegga la GUI SOCS-SI [6], è possibile avviarla direttamente dall'interfaccia di Eclipse per mezzo di un tasto (figura 10) situato, ancora una volta, sulla toolbar, il quale ci permetterà di selezionare il progetto col quale configurarla.

3.2.11 Help

Nel caso l'utente abbia bisogno di qualche chiarimento riguardo SCIFF o l'utilizzo di questo plug-in, è presente un help integrato con quello della piattaforma (figura 11).

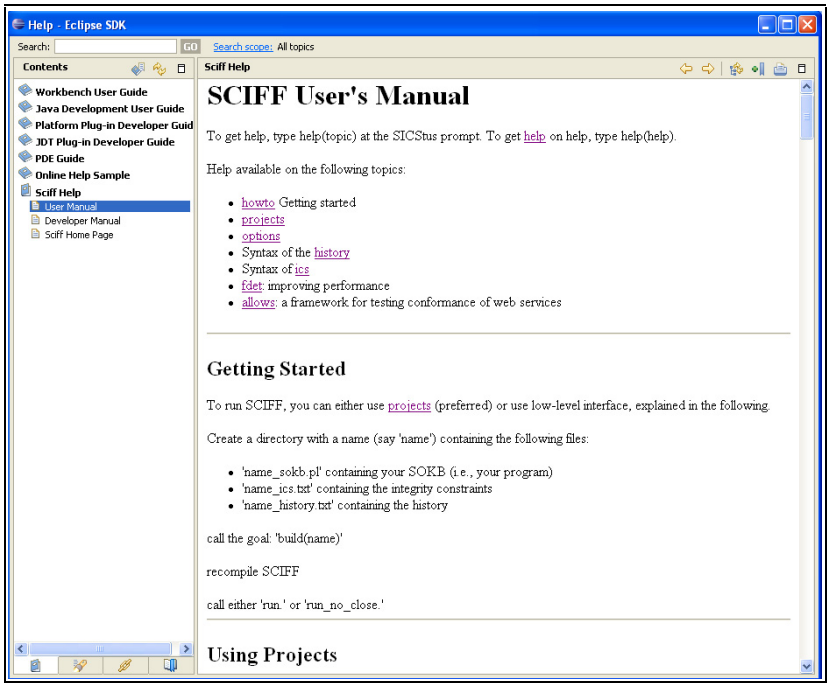

Fig. 11: SCIFF Help

Questo è consultabile accedendo alla voce “Help Contents” del menù “Help”. Questo permetterà all'utente di accedere ad una finestra contenente, sulla sinistra, una serie di manuali, tra i quali potremo consultare quello denominato “SCIFF help”.

Sulla destra verra mostrato il contenuto delle pagine dell'help selezionate, il quale, potrà essere anche stampato utilizzando, nella finestra, l'apposito tasto in alto a destra.

4 Implementazione

4.1 SCIFF preference page

Per permettere all'utente di salvare dei dati necessari al plugin (come ad esempio i direttori di installazione delle applicazioni da utilizzare), in modo permanente tra una sessione di lavoro e l'altra, Eclipse mette a disposizione un oggetto apposito. Questo potrà essere ottenuto facendone richiesta all'istanza della classe che rappresenterà il plugin in uso (in questo caso SCIFFPlugin). Per renderlo facilmente accessibile anche all'utente, però, sarà utile potervi interagire attraverso una finestra con appositi controlli per l'acquisizione dei dati. Proprio per questo, nel menù "Window"->"Preferences..." troveremo una finestra che Eclipse adibisce al salvataggio delle preferenze da parte dei plugin della piattaforma.

Per contribuire alla finestra in questione aggiungendovi una pagina dedicata alle preferenze di SCIFF, occorre in prima battuta estendere l'extension point individuato dalla stringa `org.eclipse.ui.preferencePages`, creando, all'interno della sua dichiarazione nel file `plugin.xml`, un nuovo elemento `page`. Quest'ultimo richiederà di implementare una classe che permetterà di creare la pagina di preferenze di SCIFF rappresentata dalla classe `sCIFF.SciffPathPreferencePage`. Quest'ultima, oltre all'acquisizione dei dati inseriti dall'utente, effettuerà, tramite il metodo statico `Utility.refreshTocFile`, l'aggiornamento del file che punta alle pagine dell'help di cui si parlerà nel paragrafo 4.12 .

4.2 Ics, sokb, history e SCIFF project content type

Un problema che ci si è posti all'inizio dell'implementazione è stato il cercare di capire quanti e quali tipi diversi di file avrebbe dovuto manipolare l'utente finale. Questi saranno i medesimi che vengono utilizzati da SCIFF: ics, sokb, history ed un file progetto di SCIFF denominato "project.pl".

Per sottolineare questa differenziazione, si è sfruttato un extension point messo

a disposizione dalla piattaforma ed identificato dalla stringa `org.eclipse.core.runtime.contentTypes` .

Questo permette, una volta definito, di indicare ad Eclipse il particolare tipo di contenuto presente in un file. Esistono tre modi, in Eclipse, previsti per associare un content type ad un file:

- tramite un elenco di estensioni
- elenco di nomi di file (comprendenti le estensioni)
- tramite una classe implementata dallo sviluppatore e dichiarata nell'attributo "describer" che valuta il contenuto del file

Per i tipi di file ics, sokb ed history, in questo caso, si è scelta l'associazione per estensione, mentre nel caso del file `project.pl`, quella per nome.

4.3 SCIFF nature

Un altro problema che si è cercato di risolvere è stato quello di rendere immediatamente distinguibile, agli occhi della piattaforma, un progetto SCIFF da uno di qualsiasi altro tipo. Questo è stato possibile sfruttando il concetto di "project nature"[7] che Eclipse mette a disposizione.

Espandendo l'extension point `org.eclipse.core.resources.natures` all'interno del file plugin.xml, infatti, si rende visibile alla piattaforma una nuova tipologia di "nature". Questo non è altro che un attributo che, se assegnato ad un progetto presente nel workspace, permetterà alla piattaforma di distinguerlo dagli altri.

```
<extension
      id="SCIFFnature"
      point="org.eclipse.core.resources.natures">
   <content-type id="SCIFF.project"/>
   <content-type id="SCIFF.IcsType"/>
   <content-type id="SCIFF.HisType"/>
   <content-type id="SCIFF.SokbType"/>
</extension>
```

All'interno dell'elemento che definisce l'extension point, è stato poi possibile indicare una serie di elementi `content type`, i quali risultano così collegati alla nature in questione, stabilendo una sorta di affinità tra i due. Questo vuol dire

che, nel caso di un conflitto nell'assegnazione di un content type, verrà privilegiato quello affine alla data nature.

In aggiunta è stato possibile evidenziare visivamente tutti i progetti appartenenti alla stessa nature tramite l'estensione dell'extension point `org.eclipse.ui.ide.projectNatureImages`, il quale ci ha permesso di inserire un'icona apposita.

4.4 Navigator view

Successivamente si è voluto permettere all'utente finale di poter visualizzare solo i progetti di tipo SCIFF, creati o importati tramite questo plugin. Ciò lo si è potuto fare sfruttando un componente già presente nella piattaforma e fornito tramite la classe `ResourceNavigator`. Questa rappresenta una view che visualizza, tramite una struttura ad albero, le risorse contenute nel workspace di Eclipse. Nel caso si fosse adottata, però, la classe originale, si sarebbe avuta una visualizzazione di tutti i contenuti del workspace e non solamente di quelli di maggiore interesse per lo sviluppatore SCIFF, ovvero i progetti che presentano una SCIFF nature.

Per permettere ciò, infatti, si è proceduto all'estensione della classe `ResourceNavigator`, la quale presenta un meccanismo di filtraggio delle risorse presenti nel workspace. Si è scelto quindi di procedere

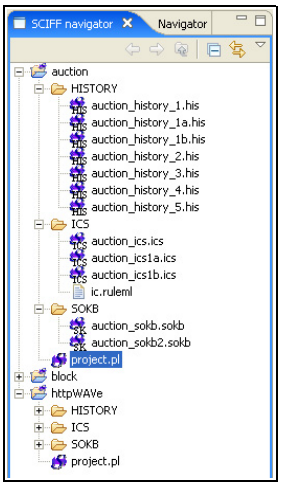

Fig. 12: SCIFF navigator

sovrascrivendo il metodo di inizializzazione dei filtri `initFilters`, creando quindi l'istanza di un filtro progettato su misura. Quest'ultimo, presente nella classe `SciffProjectFilter`, discerne tra le risorse del workspace tramite il metodo `select`, del quale riportiamo il codice.

```
public boolean select(Viewer viewer, Object parentElement, Object element){

            try{
                  if(element instanceof IProject)
                  {
                  IProject project=((IResource)element).getProject();
                  if(project.isNatureEnabled("SCIFF.SCIFFnature") )
                  super.select(viewer,parentElement,element);
                  else return false;
                  }
            }catch(CoreException e){return false;}
        return super.select(viewer,parentElement,element);
    }
```

In fine si è provveduto all'integrazione col workbench della view tramite l'estensione dell'extension point `org.eclipse.ui.views` nel file plugin.xml, e, in particolare, alla dichiarazione della classe `sCIFF.NavigatorView` in (Fig.12) nell'attributo `class` nell'elemento view.

4.5 Sciff Perspective

Dal momento che ogni plugin installato nella piattaforma porta un suo contributo anche in termini di elementi grafici all'interno dell'interfaccia, si potrà facilmente arrivare ad un eccessivo numero di questi ultimi, creando così, talvolta, confusione nell'utente. Per questo motivo si è avvertita l'esigenza di effettuarne una selezione, discernendo, in particolare, quelli di cui l'utente avrebbe fatto uso nello sviluppo del proprio progetto di SCIFF. A tal scopo, ci viene incontro uno strumento di Eclipse denominato “perspective”, di cui si è discusso nel paragrafo 2.5. Per aggiungere alla piattaforma una perspective da noi implementata, si è proceduto all'estensione dell'extension point `org.eclipse.ui.perspectives` come segue:

```
<extension
         point="org.eclipse.ui.perspectives">
      <perspective
            class="sCIFF.PerspectiveFactory"
            id="SCIFF.perspective"
```

```
            name="SCIFF perspective"/>
   </extension>
```

Come possiamo vedere, con l'attributo `class` si è indicato ad Eclipse la classe che implementa l'attivazione della perspective vera e propria.

Questa classe, implementando l'interfaccia `IPerspectiveFactory`, renderà disponibile il metodo `createInitialLayout`. Questo si incaricherà di fissare la struttura iniziale della perspective, stabilendo le posizioni iniziali, quante e quali finestre verranno visualizzate.

In questo caso, si è deciso di inserire all'apertura della perspective (Fig. 13):

(1) sul lato sinistro della schermata, le finestre "Navigator" e "Sciff Navigator"

(2) nella parte inferiore la finestra "Problems"

(3) la parte restante è stata tenuta vuota, per riservarla alle finestre degli editor che verranno aperte durante la sessione di lavoro dell'utente.

Come spiegato in precedenza, la finestra "Sciff Navigator" ci permetterà di visualizzare tutti e soli i progetti che presenteranno la "Sciff nature", la finestra "Navigator", invece, ci permetterà di visualizzare l'intero contenuto del workspace di Eclipse. Infine la finestra "Problems" ci consentirà di tenere sotto controllo i problemi che si riscontreranno in fase di lancio dei progetti.

4.6 Editor

Per lo sviluppo degli editor implementati si è deciso, per lo più, di farlo estendendo la classe `TextEditor` già presente nelle librerie di Eclipse. Quello che ha spinto in tale direzione è stata la possibilità di sfruttare meccanismi, in essa implementati, atti a facilitare le più comuni operazioni messe a disposizione da un editor, come ad esempio l'evidenziazione della sintassi, oppure l'attivazione di suggerimenti al programmatore sui contenuti da inserire.

Per meglio comprendere l'implementazione effettuata negli specifici casi, può essere utile osservare la struttura della classe di base `TextEditor`.

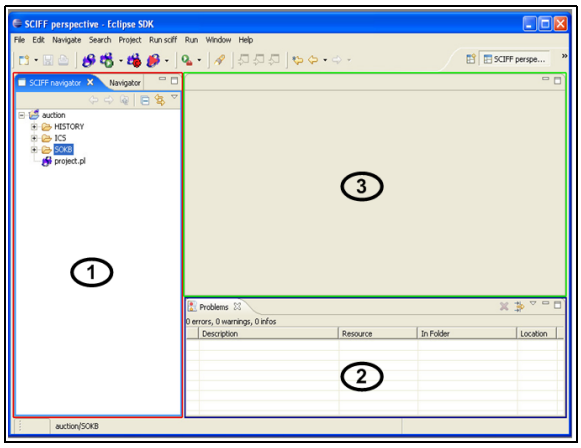

Fig. 13: SCIFF perspective

4.6.1 La classe TextEditor

E' importante sottolineare il fatto che gli editor di testo, in Eclipse, adottano un disaccoppiamento tra contenuti e loro presentazione.

Al momento dell'apertura di un file, l'editor ad esso associato riceverà in ingresso dalla piattaforma un oggetto `IEditorInput` (Fig. 14). Questo descriverà la risorsa in procinto di essere editata tramite le sue caratteristiche di base (ad esempio evidenzierà se la risorsa è un file oppure uno stream di dati).

A questo punto, l'editor invierà l'`IEditorInput` all'oggetto `IDocumentProvider`, il quale gli restituirà il relativo oggetto `IDocument`. Sarà quest'ultimo l'oggetto sul quale l'utente andrà ad effettuare la vera e propria fase di editing.

Per quanto riguarda la formattazione vera e propria del testo all'interno dell'editor, invece, sarà svolta dall'oggetto `TextViewer` il quale si occuperà di gestire i dettagli di più basso livello riguardanti il lato di presentazione. Questo, infatti, farà corrispondere il tipo di formattazione specifico per una zona dell'oggetto `IDocument`.

Questa corrispondenza verrà specificata dallo sviluppatore in una classe che

estenderà `SourceViewerConfiguration`, la quale sarà passata all'editor tramite il metodo `setSourceViewerConfiguration`.

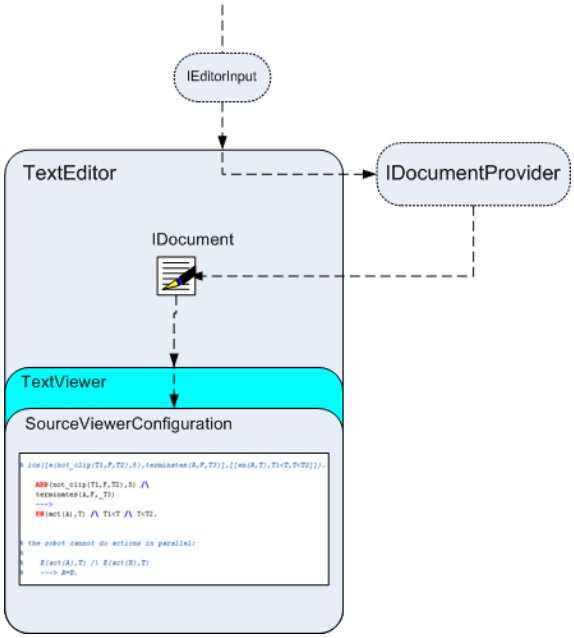

Fig. 14: TextEditor

4.6.2 ICS editor

L'editor di file ICS, istanziato dalla classe `IcsEditor`, è, come già accennato, un'estensione della classe `TextEditor`, e come tale ne erediterà le strutture e i meccanismi interni. Proprio per questo, al fine di personalizzare la presentazione del documento, è stata implementata la classe `IcsEditorConfiguration` estendendo `SourceViewerConfiguration`.

Le funzionalità implementate, come detto nei capitoli precedenti, sono essenzialmente di tre tipi: un meccanismo di syntax highlighting, una evidenziazione del bilanciamento delle parentesi ed una guida, sotto forma di finestra popup, sui parametri che l'utente andrà ad inserire all'interno dei predicati.

Per quanto riguarda il meccanismo di syntax highlighting, esso segue il modello detto “damage, repair and reconciling”.

La fase di “damage” avverrà quando l'utente si troverà ad editare il documento, e la sua gestione sarà affidata all'oggetto `IPresentationDamager`. La fase di

“repair”, invece, sarà gestita dall'oggetto `IPresentationRepairer`. Questa consisterà nelle modifiche da effettuare sul documento in precedenza editato dall'utente, per riportare la presentazione in una situazione di coerenza con i nuovi contenuti. La fase di “reconciling” invece (gestita dall'oggetto `IPresentationReconciler`), consisterà in una gestione e supervisione degli oggetti `IpresentationDamager` e `IpresentationRepairer`.

```
public IPresentationReconciler getPresentationReconciler(ISourceViewer
sourceViewer)
{
	PresentationReconciler reconciler=new PresentationReconciler();
	DefaultDamagerRepairer dr=new DefaultDamagerRepairer(getScanner());
	reconciler.setDamager(dr,IDocument.DEFAULT_CONTENT_TYPE);
	reconciler.setRepairer(dr,IDocument.DEFAULT_CONTENT_TYPE);

	return reconciler;
}
```

Nel caso specifico di questo editor, viene creato un oggetto per ciascuna delle fasi di damage. repair e reconciling. L'ultimo in particolare, è stato creato passando come parametro uno oggetto `ITokenScanner`, fornito dal metodo `getScanner`, che effettuerà una scansione del testo editato e si occuperà di assegnargli, in base a regole specifiche, un oggetto `Token` all'interno del quale si troveranno le specifiche di formattazione del testo.

Per quanto riguarda invece la guida sui parametri da inserire all'interno dei predicati utilizzati negli ICS, si è sfruttato anche qui una funzionalità ereditata dalla classe `TextEditor` al fine di fornire suggerimenti contestualizzati all'utente sul codice da inserire. La logica di visualizzazione dei suggerimenti sarà specificate all'interno di una classe che dovrà implementare l'interfaccia `IContentAssistProcessor`, in questo caso SciffContentAssistProcessor, mentre il controllo sull'opportunità di continuare a mantenere visualizzate le informazioni contestuali, anche dopo eventuali modifiche da parte dell'utente, è affidato alla classe `EventInformationValidator`.

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer)
{
	ContentAssistant assistant =new ContentAssistant();
	ContextInformation infoContext=new
ContextInformation(IDocument.DEFAULT_CONTENT_TYPE,"(Message,Time)");
	SciffContentAssistProcessor processor=new
SciffContentAssistProcessor(infoContext);
	EventInformationValidator validator=new EventInformationValidator();
```

```
        processor.setContextInformationValidator(validator);
        assistant.setContentAssistProcessor(processor,
                                        IDocument.DEFAULT_CONTENT_TYPE);
        assistant.enableAutoActivation(true);
        assistant.setAutoActivationDelay(300);
        assistant.setContextInformationPopupOrientation(IContentAssistant.
CONTEXT_INFO_ABOVE);
        return assistant;
}
```

4.6.3 Sokb e history editor

Per quanto rigurda questi due editor rappresentati rispettivamente dalle classi SokbEditor e HistoryEditor, mantengono pressochè intatta la logica di implementazione del precedente editor con la sola differenziazione data dal diverso tipo di sintassi da mettere in evidenza.

```
1   :- dynamic ics_file/1, sokb_file/1, history_file/1, required_option/2.

    history_file('/HISTORY/block_history.his').
    ics_file('/ICS/block_ics.ics').

2   sokb_file('/SOKB/block_sokb.sokb').

    required_option(seq_act,on).
    required_option(factoring,on).

    %%%%%%%%%%%%%%%%%%%%%%% Constant Part %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    build_prj(Path):-
        findall(F,ics_file(F),ICS_files), append_path(Path,ICS_files,IcsPathFiles),
        translate_ics_files(IcsPathFiles,'./ics.pl'),
        findall(F,history_file(F),Hist_files),  append_path(Path,Hist_files,HistPathFiles),
3       translate_histories(HistPathFiles,'./history.pl'),
        findall(F,sokb_file(F),Sokb_files),     append_path(Path,Sokb_files,SokbPathFiles),
        convert_sokb(SokbPathFiles,'./sokb.pl'),
        compile(sokb), compile(history), compile(ics),
        findall([O,V],required_option(O,V),LOptions),
        set_options(LOptions).

4   run(_):- iter.

    % Default:
    %run(_):- run.
    %run_open(_):- run_no_close.
    %run_closed(_):- run.
```

Fig. 15: Struttura file project.pl

4.6.4 Project editor

E' utile, innanzitutto, menzionare brevemente la struttura del file project.pl (Fig. 15). Questo è formato:

1. innanzitutto da una intestazione che rimane fissa;
2. da una parte di dichiarazione dei file e delle opzioni che andranno a contribuire al progetto;
3. da un'ulteriore parte che rimane costante;
4. da una quarta parte all'interno della quale possiamo trovare le definizioni delle modalità di lancio di un progetto SCIFF (ovvero: `run(_)`, `run_closed(_)`, `run_open(_)`).

In particolare, all'interno della seconda parte, per attivare un'opzione si affiungerà il fatto prolog "`required_option(nome_opzione,valore).`" , mentre per quanto riguarda l'aggiunta di un file, la sintassi cambierà a seconda del tipo che si vorrà aggiungere. Per i file ics infatti sarà "`ics_file('nome_file').`", per i file sokb "`sokb_file('nome_file').`" e per i file history sarà "`history_file('nome_file').`".

Per semplificare l'inserimento di dati nel file project.pl, si è scelto di inserire, in aggiunta ad un editor testuale convenzionale, un'interfaccia a checkbox. Si è pensato, quindi, che il sistema migliore per mettere assieme questi due aspetti in modo uniforme col workbench, fosse quello di implementare il tutto sotto forma di un editor multi pagina.

Per fare ciò si è proceduto alla creazione di un'estensione della classe `MultiPageEditorPart`, la quale ci ha permesso di aggiungere varie pagine accessibili in un'unica finestra di editor. Questo avverrà tramite il metodo `createPages`, il quale, a sua volta, ne richiamerà alti due: `createControlPage` e `createEditorPage`. Quest'ultimo non farà altro che istanziare una classe `ProjectTextEditor` (estensione di `TextEditor`) che verrà aggiunta come pagina del nuovo editor tramite il metodo `addPage`, mentre il primo istanzierà la pagina contenente i controlli checkbox.

E' importante sottolineare come i checkbox non modificano il contenuto del file project.pl, bensì quello dell'oggetto `IDocument` gestito dall'istanza della classe `ProjectTextEditor`. Per avere la modifica sul file, l'utente dovrà salvare la modifica.

Il metodo `createControlPage` provvederà ad una chiamata al metodo `getCheckFileResources`, il quale popolerà gli array `sokbCheck`, `icsCheck`, e `historyCheck`, con i file sokb, ics e history presenti nel progetto attualmente selezionato. La discriminazione sull'array di destinazione del file, sarà poi effettuata in base all'estensione dello stesso. Questi array conterranno oggetti di tipo `CheckFile`, i quali vedranno al loro interno sia l'istanza di un checkbox

che l'oggetto `IFile` ad esso corrispondente. Una volta istanziati tutti i checkbox, viene effettuato un controllo dell'esistenza del path, relativo alla directory di progetto, all'interno del file project.pl per verificarne lo stato di selezione. In aggiunta sono applicati degli ascoltatori di eventi di selezione, i quali provvederanno a verificare un'eventuale cambiamento dello stato di selezione dei controlli. In caso di modifica, si procede ad effettuare le corrispondenti modifiche attraverso i metodi statici della classe `Project`.

Ad esempio: nel caso in cui l'utente selezioni un checkbox con etichetta "`DIRECTORY/nome_file.ics`", in quel caso si provvederà ad aggiungere sull'oggetto `IDocument` della finestra "Text Editor" la stringa "`ics_file('/DIRECTORY/nome_file.ics').`".

Al momento del salvataggio dei contenuti, verrà invocato il metodo doSave, il quale non farà altro che richiamare lo stesso metodo nell'istanza della classe TextEditor. Sarà quindi solo la pagina "Text Editor" ad effettuare veramente il salvataggio del file.

4.7 Wizard

Per implementare un nuovo wizard in Eclipse, si procede, prima di tutto, alla dichiarazione nel file `plugin.xml` del relativo extension point, aggiungendovi all'interno la dichiarazione di un nuovo elemento `wizard`.

Per ognuno di questi vanno poi definiti vari attributi tra i quali l'attributo `class` che dichiara quale classe (estensione della classe `Wizard`) si occuperà della gestione della finestra di wizard.

Per aggiungere una nuova pagina al wizard, inoltre, si procede istanziando una nuova classe `WizardPage`, la quale, attraverso il proprio metodo `createControl`, potrà specificare i controlli in essa presenti.

4.7.1 New SCIFF project e SCIFF file wizard

Per permettere all'utente la creazione, in modo facile e veloce, di un nuovo

progetto SCIFF o un nuovo file ics, sokb oppure history, si è pensato di implementare queste funzionalità all'interno di due wizard.

In questo caso si è dichiarata l'estensione dell'extension point `org.eclipse.ui.newWizards`, all'interno del quale sono stati aggiunti i quattro elementi wizard identificati dalle stringhe `SCIFF.newSCIFF`, `SCIFF.newSokb`, `SCIFF.newIcs` e `SCIFF.newHis`.

```
<extension
         point="org.eclipse.ui.newWizards">
      <wizard
            class="sCIFF.NewSCIFF"
            finalPerspective="SCIFF.perspective"
            icon="icons/sciff icona.gif"
            id="SCIFF.newSCIFF"
            name="Project"
            preferredPerspectives="SCIFF.perspective"
            project="true"/>
      <wizard
            class="sCIFF.NewSokb"
            icon="icons/sciff icona.gif"
            id="SCIFF.newSokb"
            name="Sokb file"
            project="false"/>
           ........
   </extension>
```

Per ognuno di questi, infine, si è specificata la classe che andrà ad istanziare il wizard vero e proprio, ovvero: `sCIFF.NewSCIFF`, `sCIFF.NewSokb`, `sCIFF.NewIcs` e `sCIFF.NewHistory`.

Nel caso del wizard per la creazione di un nuovo progetto, si sono dichiarati anche altri due attributi: `finalPerspective` permette di indicare quale perspective aprire dopo la creazione del progetto, mentre `preferredPerspective`, permette, nel caso la perspective qui specificata sia già aperta in Eclipse, di non aprirla una seconda volta.

4.7.2 Export Wizard

Per aggiungere la possibilità all'utente di esportare i propri progetti SCIFF in formato RuleML velocemente, si è scelto di adottare il formato di export wizard.

Per fare questo si è esteso l'extension point `org.eclipse.ui.exportWizards`, all'interno del quale è stato aggiunto l'elemento wizard identificato dalla stringa `SCIFF.exportRulemlwizard`. Attraverso il suo attributo `class`, si è poi provveduto a specificare la classe `ExportRulemlWizard` che si occuperà di creare l'istanza del wizard, mentre la classe `ExportRulemlPage` ne istanzierà la pagina.

In questo caso, all'interno del metodo `createControl` dell'oggetto `ExportRulemlPage`, troviamo un oggetto `ComboBox` che permette all'utente una scelta rapida del progetto da esportare. In particolare, si è cercato di inserire nella lista del `ComboBox` solamente quei progetti che presentavano la caratteristica "SCIFF nature", usando il metodo `Utility.getSciffNature`, che opera passando in rassegna tutti i progetti presenti nel workspace.

Per facilitare l'indicazione da parte dell'utente del direttorio di destinazione, si è anche provveduto ad inserire un oggetto `Button` nell'interfaccia. A `Button` è applicato un ascoltatore di eventi di selezione `SelectionListener` (in questo caso si è adoperato la versione `SelectionAdapter` data la necessità di implementare solo un metodo tra tutti), programmato per rispondere ad un'eventuale selezione del pulsante con la creazione di un'istanza della classe `DirectoryDialog`, che crea una finestra di selezione della directory desiderata.

```
containerBrowseButton.addSelectionListener(new SelectionAdapter() {
 public void widgetSelected(SelectionEvent event)
             {
                DirectoryDialog dialog=new
             DirectoryDialog(containerNameField.getShell());
                dialog.setMessage("Seleziona directory origine:");
                String selectedDirectory = dialog.open();
                if (selectedDirectory != null)
                    {
                     containerNameField.setText(selectedDirectory);
                    }
             }
});
```

Si noti, inoltre, che su ogni istanza della classe `Text` e della classe `ComboBox` si è attivato un ascoltatore di eventi (eventi di modifica del campo di testo, e di selezione nel caso del `ComboBox`). Questi hanno il compito di controllare lo stato dei campi di inserimento dati e di verificarne il popolamento collettivo.

Nel caso la verifica dia esito positivo, verrà richiamato il metodo `setPageComplete(true)` della classe che istanzia la pagina del wizard, attivando così il pulsante "Finish" sul fondo dell'ultima finestra.

La pressione del pulsante effettuerà la chiamata al metodo `exportToRuleml` dell'istanza della classe `ExportRulemlPage`, il quale inizia e porta a termine la procedura di esportazione effettiva del progetto. La classe crea un nuovo processo che esegue prima SICStus Prolog, carica l'interprete SCIFF, ed invoca il goal "`ruleml_parser:save_ics_ruleml('file di destinazione').`". All'interno di quest'ultimo, in particolare, viene passato il parametro che rappresenta il file di destinazione compreso di direttorio assoluto. In questo caso, si è sfruttata una modalità di lancio di SICStus Prolog che ci permette, passando all'avvio il parametro `--goal`, di indicare una lista di goal che dovrà eseguire una volta partita la sessione del programma.

```
public boolean exportToRuleml()
{
try{
getWizard().getContainer().run(false,false,new IRunnableWithProgress()
       {
       public void run(IProgressMonitor monitor)
              {
              . . . . . .
              String[] cmdLine={sicstusPath+"\\sicstus.exe",
"--goal","['"+sciffPath+"/sciff.pl'],project("+projectNameValue+"),
ruleml_parser:save_ics_ruleml('"+destPath+"/"+fileNameFieldValue+".ruleml').
"};
              monitor.beginTask("SCIFF export",20);
              Process p=Runtime.getRuntime().exec(cmdLine,null,sicstusDir);
              LineNumberReader stream=new LineNumberReader(new
       InputStreamReader(new BufferedInputStream(p.getErrorStream())));

              while (!prova.matches("(?s).*consulted"
              +sicstusMatch+ "/ics.pl in module ics.*"))
              {
              prova=stream.readLine();
              if (prova.matches("(?s:.*consulted
              "+sciffPath+"/sciff.pl in module user.*)"))
                     {
                     monitor.worked(10);
                     }
              if(prova.startsWith("!")||prova.matches(".*goal failed.*"))
                     {
                     Utility.showError("Project exporting failed!",false);
                     break;
                     }
              }
              monitor.done();
              stream.close();
              p.destroy();
              }
}
```

Dati i tempi che una tale operazione potrebbe comportare, si è optato per inserire nell'interfaccia una barra che rappresentasse l'avanzamento dell'attività. Per fare questo, nella fase di inizializzazione della classe `ExportRulemlWizard` si è eseguito il metodo `setNeedsProgressMonitor(true)` per far sì che venga riservato un apposito spazio nell'interfaccia. Il processo da monitorare è stato così implementato all'interno del metodo `run` dell'interfaccia `IrunnableWithProgress` passata come parametro al metodo `run` dell'oggetto `Container` della pagina di wizard. Una volta fatto partire il processo, si è proceduto al controllo dello stream in uscita da quest'ultimo, attendendo il messaggio indice della riuscita dell'operazione e ricercando un eventuale messaggio di errore. In caso di errore, viene inviato all'utente un messaggio di errore tramite una finestra, poi, in entrambi i casi, si termina il processo.

4.7.3 Import SCIFF project

Per aggiungere un wizard che importa all'interno del workspace un progetto di SCIFF preesistente, si è esteso l'extension point `org.eclipse.ui.importWizards`. In seguito si è creata una classe `ImportSCIFF`, che si occuperà di rappresentare l'istanza del wizard, e la classe `SCIFFImportPage`, che si occuperà di rappresentare l'istanza della pagina del wizard.

Analogamente al caso precedente, all'interno del metodo `createControl` della classe `SCIFFImportPage`, oltre agli altri elementi dell'interfaccia, è stato inserito un oggetto `Button` che ci permetterà di istanziare la classe `DirectoryDialog` per facilitare la scelta del direttorio di origine da parte dell'utente.

Una volta avvenuta la fase di inserimento dati, premendo il pulsante “Finish”, analogamente a quanto spiegato nel paragrafo precedente, verrà invocato il metodo `performFinish` della classe `ImportSciff`, che a sua volta invocherà il metodo `createProject` appartenente all'istanza della classe `SCIFFImportPage`.

Questo metodo si preoccuperà di:

- creare uno stream per la copia del contenuto del file project.pl di origine
- creare, con l'ausilio del metodo statico `CleanSCIFFproject.CreateImport`, la struttura di base del progetto di SCIFF. Questo agirà in modo analogo al metodo `CleanSCIFFproject.Create` tralasciando però la creazione del file project.pl
- infine viene chiamato il metodo `copyFolderContent` che ha come fine l'importazione dei file sorgente all'interno della struttura del progetto precedentemente creata

Questo metodo prima cattura il contenuto della copia appena effettuata del file project.pl in una stringa, poi richiama il metodo `scanDirectory`. In ultima fase, il metodo `copyFolderContent` procederà all'aggiornamento del file project.pl tramite il metodo `updateProjectFile`.

Analizzando il metodo `scanDirectory`, si può vedere come, dopo aver inserito in un array tutti gli oggetti `File` trovati nel direttorio di origine `dir`, li si esamina uno per volta.

Nel caso l'oggetto `File` preso in considerazione sia in realtà una directory, questa verrà creata con lo stesso nome all'interno della directory di destinazione, e, successivamente, si provvederà a chiamare ricorsivamente lo stesso metodo `scanDirectory`.

Nel caso in cui, invece, l'oggetto `File` rappresenti un vero e proprio file, il suo nome verrà analizzato per individuarne il tipo.

Ad esempio, se la stringa “ics” sarà presente all'interno del nome del file, questo verrà copiato all'interno della directory “ICS”, appositamente creata dal metodo `CleanSCIFFproject.CreateImport` per i file di tipo ICS. Al contrario, nel caso in cui nel nome non sia stato riscontrato alcun riferimento ad un particolare tipo di file usato da SCIFF, il file in questione sarà copiato

nell'attuale directory di destinazione indicata dall'oggetto `dest` ricevuto come parametro dal metodo in esame.

```
private void scanDirectory(File dir,IContainer dest)
throws IOException{
try{
File[] files=dir.listFiles();
for(int i=0;i<files.length;i++)
{
      .........

if(files[i].isDirectory())
      {
      Path path=new Path(files[i].getName());
      IFolder folder=dest.getFolder(path);
      folder.create(IFolder.FORCE,true,null);
      scanDirectory(files[i],folder);
      }
else
{
if(files[i].isFile())
      {
      .........
if(files[i].getName().matches(".*ics.*"))
      {
      IFolder destFolder=dest.getProject().getFolder("ICS");
      Utility.copy(files[i],destFolder,"ics");
      String filePath=Utility.getRelativePath(files[i],dir);
      filePath=Project.getPathToMatch(filePath);
      if (projectContent.matches("(?sm:.*"+filePath+".*)"))
            {
String renamedExtension=Utility.renameExtension(files[i].getName(),"ics");
projectContent=projectContent.replaceAll(".*"+filePath+".*","ics_file('/ICS/
"+renamedExtension+"').");
            }
      }else
      .........
if(!files[i].getName().equals("project.pl"))
      {
      Utility.copy(files[i],dest,"other");

      }
      }
      .........
}
```

4.7.4 Import SCIFF file

Dal momento che, anche in questo caso, si è avuto a che fare con un wizard di importazione, l'extension point esteso è stato il medesimo. La classe che rappresenta l'intero wizard, invece, è `ImportSCIFFfile`, mentre, la classe `FileImportPage` gestisce le singola pagina.

L'utente, quando sceglie sotto forma di quale tipologia di file importare la risorsa sorgente, non potrà effettuare una scelta multipla, bensì l'una escluderà

le altre. A questo tipo di scelte si adegua perfettamente la classe `Button` istanziata specificando nel parametro `type` il valore `SWT.RADIO`. Questo oggetto creerà un radio button, consentendo per sua natura di effettuare una sola scelta per volta.

A questo elemento, in particolare, è applicato un ascoltatore di eventi che, se avvertirà un cambiamento di selezione del radio button, associerà l'etichetta di quest'ultimo alla variabile `radioStatus`.

Il pulsante "Finish" richiamerà il metodo `importFile`, il quale, a seconda del valore della stringa `radioStatus`, deciderà in che modo importare il file.

```
public boolean importFile()
{
String project=projectCombo.getItem(projectCombo.getSelectionIndex());
this.project=ResourcesPlugin.getWorkspace().getRoot().getProject(project);
File file=new File(containerNameField.getText());
try{
FileInputStream fileStream=new FileInputStream(file);

if(radioStatus.equals("Generic"))
       {
       IFile fileOut=this.project.getFile(file.getName());
       fileOut.create(fileStream,true,null);
       return true;
       }
if(radioStatus.equals("Ics"))
       {
       IFolder dir=this.project.getFolder("ICS");
       if (dir.exists())
              {
              String newName=Utility.renameExtension(file.getName(),"ics");
              IFile fileOut=dir.getFile(newName);
              fileOut.create(fileStream,true,null);
              return true;
              }
              else return false;
       }
              ..............

}catch(FileNotFoundException e){return false;}
catch(CoreException e){return false;}
}
```

Se, ad esempio tra i radio button sarà selezionato il valore "ics", al file verrà dapprima modificata l'estensione (in questo caso ".ics") mediante il metodo statico `Utility.renameExtension`, successivamente verrà copiato il file. Nel caso invece sia selezionata la voce "Generic" il file verrà copiato nella cartella del progetto senza modificarne l'estensione.

4.8 Lancio del'interprete SCIFF

Nell'utilizzo di un ambiente di sviluppo integrato, non è rara l'esigenza da parte dell'utente di dover lanciare un'applicazione esterna in modo integrato con l'interfaccia del programma in uso. Questo può essere l'esempio di un ambiente di sviluppo per applicazioni web-based, dove l'utente si troverà spesso a dover utilizzare applicazioni server da avviare e sfruttare all'interno della piattaforma di sviluppo. Poi, nel momento in cui nasca l'esigenza di personalizzare l'avvio dell'applicazione, magari avendo la necessità di passare determinati parametri da linea di comando, si fa più forte il bisogno di disaccoppiare la fase di raccolta e gestione di dati da inviare all'applicazione dalla fase di lancio vera e propria. Per dare ascolto a questa necessità, Eclipse mette a disposizione dello sviluppatore un meccanismo che prevede proprio questo tipo di disaccoppiamento.

In particolare, lo sviluppatore utilizzerà le classi che implementano l'interfaccia `IlaunchConfiguration` come configurazioni di lancio, ovvero, oggetti all'interno dei quali si andranno a riporre i parametri utili per l'avvio dell'applicazione.

Una volta terminato l'inserimento dei dati, la configurazione `ILaunchConfiguration` verrà passata dal metodo statico `launch` della classe `DebugUITools`, all'appropriato oggetto `ILaunchConfigurationDelegate`. In particolare, l'associazione tra questo oggetto e la configurazione di lancio sarà definito dal programmatore (sempre all'interno del file `plugin.xml`) attraverso l'extension point `org.eclipse.debug.core.launchConfigurationTypes`. Infine, la classe `ILaunchConfigurationDelegate` conterrà il solo metodo `launch`, il quale sarà ora responsabile dell'esecuzione vera e propria dell'applicazione.

In questo caso, è stato deciso di far partire l'applicazione tramite un pulsante nella toolbar di Eclipse. Poiché è importante poter avviare l'interprete SCIFF tramite il goal `compile` oppure `consult`, occorrerà specificare due modalità

differenti di lancio, ma, al contempo, renderle accessibili velocemente ed in modo semplice. Per soddisfare questa esigenza si è scelto di inserire queste due procedure di lancio tramite un menù accessibile cliccando su di una freccetta a fianco di un pulsante (Fig. 7). Per fare questo, è necessario estendere l'extension point `org.eclipse.ui.actionSets`, all'interno della cui definizione verrà aggiunto un elemento `action` ogni volta che si avrà l'intenzione di aggiungere un pulsante, un menù o una sua voce al workbench.

```
<extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            id="SCIFF.actionSet"
            label="SCIFF.actionSet"
            visible="true">
         <action
               class="sCIFF.launch.StartSciffPulldown"
               disabledIcon="icons/Start sciff.gif"
               hoverIcon="icons/Start sciff.gif"
               icon="icons/Start sciff.gif"
               id="SCIFF.StartSciff"
               label="Start SCIFF "
               style="pulldown"
               toolbarPath="Normal/additions"
               tooltip="Start SCIFF"/>
          ..................
      </actionset>
</extension>
```

In questo caso, ad esempio, nell'elemento `action` identificato dalla stringa `SCIFF.StartSciff`, specificando nell'attributo `style` il valore `pulldown`, si è ottenuto proprio tale tipo di pulsante.

Attraverso l'implementazione dell'interfaccia `IWorkbenchWindowPulldownDelegate`, inoltre, potremo definire il menù e le sue voci, rappresentati rispettivamente dalle classi `Menu` e `MenuItem` all'interno del metodo `getMenu`.

Agli oggetti `MenuItem`, poi, sono stati aggiunti ascoltatori di eventi di selezione, i quali reagiranno con la chiamata ad uno dei metodi `startConsult` o `startCompile`, a seconda della voce del menù coinvolta nella selezione.

Questi due metodi risultano praticamente identici ai fini dell'implementazione, differenziandosi soltanto per i parametri passati in fase di creazione del

processo.

```
public void startCompile()
{
boolean found=false;
ILaunchConfiguration configuration;
ILaunchManager manager=DebugPlugin.getDefault().getLaunchManager();
ILaunchConfigurationType type=manager
      .getLaunchConfigurationType("SCIFF.launchSciffConfigurationType");
try{
ILaunchConfiguration[] configurations=manager.getLaunchConfigurations(type);
for(int i=0;i<configurations.length;i++)
      {
      if (configurations[i].getName()
            .equals(LaunchSciffConfigurationDelegate.START_SCIFF_NAME))

            {
            found=true;
            configuration=configurations[i];
            ILaunchConfigurationWorkingCopy workingCopy=configuration
            .copy(LaunchSciffConfigurationDelegate.START_SCIFF_NAME);
workingCopy.setAttribute(LaunchSciffConfigurationDelegate.SCIFF_CONSULT
,"compile");
            configuration=workingCopy.doSave();
            DebugUITools.launch(configuration,"profile");
            break;
            }
            }

            if(!found)
            {
            ILaunchConfigurationWorkingCopy
            workingCopy=type.newInstance(null
                  ,LaunchSciffConfigurationDelegate.START_SCIFF_NAME);
workingCopy.setAttribute(LaunchSciffConfigurationDelegate.SCIFF_CONSULT
,"compile");
            configuration=workingCopy.doSave();
            DebugUITools.launch(configuration,"profile");
            }

      ..............
      }
```

Ad esempio, il metodo `startCompile` richiede tutte le configurazioni di lancio di tipo `SCIFF.launchSciffConfigurationType`, ovvero quelle adibite ad interagire con il processo di SCIFF, al gestore `ILaunchManager`. Questo è ottenibile tramite il metodo `getLaunchManager` della classe `DebugPlugin`. Delle configurazioni di lancio otenute in precedenza, si scelgono quelle che presentano il nome "START SCIFF", ovvero quelle incaricate di effettuare il lancio di SCIFF, e l'ultima trovata viene riutilizzata per la nuova sessione di lancio. Dal momento però, che un oggetto `ILaunchConfiguration` non dà la possibilità di essere modificato direttamente, si procederà alla creazione di un oggetto `ILaunchConfigurationWorkingCopy` copiandovi l'intero contenuto dell'oggetto `IlaunchConfiguration`. Una volta ottenuta una copia

modificabile della configurazione di lancio, si procede all'assegnamento del valore “compile” all'attributo “SCIFF consult”, il quale farà in modo che la classe che si occuperà di lanciare l'applicazione saprà di dover avviare SCIFF in modalità compile. Si è, quindi, salvato il contenuto dell'oggetto `ILaunchConfigurationWorkingCopy` nuovamente in un oggetto `ILaunchConfiguration`, il quale finalmente potrà essere inviato alla classe di lancio `LaunchSciffConfigurationDelegate` dal metodo `launch` della classe `DebugUITools`.

La classe `LaunchSciffConfigurationDelegate`, a questo punto, si occuperà di estrarre dall'oggetto `ILaunchConfiguration` il nome col quale è stato creato al fine di capire se si dovrà avviare una nuova sessione di SCIFF, o se semplicemente sarà necessario interagire con una già esistente. Nel caso di una configurazione di lancio creata col nome “START SCIFF”, avverrà una chiamata al metodo `startSciff` a cui sarà passato l'oggetto `ILaunch`, il quale rappresenterà la specifica sessione di lancio.

Il metodo `startSciff` per prima cosa richiede all'`ILaunchManager`, ottenuto dalla classe `DebugPlugin`, tutti i processi da lui gestiti, e nel caso la loro etichetta corrisponda con la stringa "`SICStus process`" si provvede a terminarli. Fatto questo si procede al recupero nel preference store delle proprietà settate dall'utente necessarie al lancio dell'applicazione. In questo caso si rendono necessarie le stringhe con indicati i direttori di installazione di SCIFF e del file `sicstus.exe`. Questi vengono poi inseriti nell'array degli argomenti passati all'eseguibile `sicstus.exe`, il quale, attraverso il parametro `-goal`, ci permetterà di effettuare subito la richiesta di compilazione del file `sciff.pl`, evitando un'ulteriore scrittura di dati nello stream di input del processo.

A questo punto, verrà eseguito il processo vero e proprio mediante la chiamata al metodo statico `exec` della classe `DebugPlugin`, il quale restituirà un oggetto `Process`. E' possibile incapsulare tale oggetto in uno del tipo IProcess, che

permetterà l'assegnazione automatica di una console per interagire col processo in modo integrato col workbench.

Una volta lanciato il processo, è stato subito evidenziato il fatto che SICStus prolog, al contrario di SCIFF, inviasse tutto l'output sullo standardError. Questo ha fatto sì che sulla console associata al processo, questi venissero visualizzati in colore rosso. Per ovviare a questo problema estetico, si è esteso l'extension point `org.eclipse.debug.ui.consoleColorProviders`, il quale, attraverso la classe `ConsoleColorProvider` specificata al suo interno, ha permesso la riassegnazione dei colori agli stream della console.

4.9 Compilazione di un progetto

Per la compilazione del progetto sviluppato dall'utente, in modo analogo a quanto fatto nel paragrafo precedente, è stato inserito nella toolbar di Eclipse un pulsante che permette un accesso immediato a questa funzionalità (Fig. 8). Come fatto precedentemente, quindi, è stato inserito nell'extension point `org.eclipse.ui.actionSets` un nuovo elemento `action` ed è stata implementata la classe `LaunchSCIFFproject`, che si incaricherà di rispondere alla selezione del pulsante nella toolbar.

Per far compilare all'interprete SCIFF un progetto, occorre, dopo aver lanciato l'interprete, eseguire il goal `project(nome-del-progetto).;` questo implicherà una scelta da parte dell'utente su quale progetto compilare. Questo problema è stato risolto con la creazione, da parte del metodo `run` della classe `LaunchSCIFFproject`, di un'istanza della classe `ProjectDialog`. Attraverso il metodo `createDialogArea` di questa classe, è stato inserito un oggetto `ComboBox` che presenta come elementi selezionabili tutti i progetti con l'attributo “SCIFF nature”.

Dal momento che la classe `ProjectDialog` sarà utilizzata non solo in fase di compilazione del progetto SCIFF, ma anche in fase di lancio dell'applicazione SOCS-SI, questà conterrà un metodo per ognuno dei due scopi. La scelta del

metodo da utilizzare sarà effettuata a seconda del valore del flag `socs_si` specificato all'atto della classe.

Nel caso in questione, all'atto della pressione del pulsante “Ok” da parte dell'utente, viene invocato il metodo `okPressed`, il quale, a sua volta invoca il metodo `compileSciffProject`. Questo metodo, in particolare, risulta analogo a quanto implementato nel metodo `startCompile` della classe `StartSciffPulldown` riportato nel paragrafo precedente, fatta eccezione per un attributo inserito nella configurazione di lancio, rappresenta il nome del progetto da compilare, e l'etichetta della configurazione che, attraverso il valore "compile_project", indica la sessione di compilazione del progetto.

Una volta passata la configurazione di lancio alla classe `LaunchSciffConfigurationDelegate`, essa ne verificherà l'etichetta, provvedendo così ad invocare il metodo più consono (in questo caso il metodo `compileProject`). Quest'ultimo controlla prima di tutto se l'interprete SCIFF è già stato lanciato, e, successivamente, viene aggiunto un'ascoltatore di eventi agli stream collegati allo standardError ed allo standardOutput del processo. Questo ascoltatore comunica quando viene aggiunta una nuova stringa alla console.

Tramite questi ascoltatori di eventi, infatti, si potrà verificare la presenza di un eventuale messaggio di errore, e, nel qual caso, provvedere all'applicazione di un marker al file nel quale questo errore si è verificato.

```
outMonitor.addListener(outListener=new IStreamListener()
{
public void streamAppended(String text, IStreamMonitor monitor)
       {
       if ((text.matches("(?sm).*[***].*"))&&
(!text.matches("(?sm).*[****] Warning[:] atom in head [***].*")))
              {
                     .............
                     getMarker(text);
                     .............
              }
       ......................
       }
});
```

Un marker è utilizzato quando si ha la necessità di annotare specifiche informazioni all'interno di un elemento del workspace senza modificarne il

contenuto. Questi possono essere di vario tipo, fra i quali possiamo trovare quelli atti a essere visualizzati all'interno della problem view per indicare, ad esempio, un errore di sintassi.

In questo caso, il metodo `getMarker` si occuperà di estrapolare, dal testo appena appeso all'output della console, i dati necessari alla creazione del marker, quali: numero della riga dell'errore oppure file in cui l'errore è stato individuato. Nel caso, ad esempio, di un errore di sintassi all'interno di un file ICS, il messaggio che compare sulla console fa riferimento al numero del vincolo all'interno del quale di troverà l'errore. Analizzando il documento di testo, dovremo quindi ricavare la posizione esatta dell'errore in termini di distanza in caratteri dall'inizio del file; questa funzione viene svolta in questo caso dal metodo statico `Utility.getLineOfIc`.

4.10 Esecuzione di un progetto

Considerando la possibilità di eseguire un progetto in tre modalità differenti offerta da SCIFF, anche questa volta si è optato per un menù di scelta accessibile tramite una freccia al fianco di un pulsante (Fig. 9). Anche per questo motivo, la sua implementazione risulta analoga a quanto trattato nel paragrafo 4.8.

In questo caso, il nome della configurazione di lancio sarà `run_project`, mentre il metodo che eseguirà il tutto si chiamerà `runProject`.

Questo metodo, in particolare, si occuperà di recuperare l'oggetto `IProcess` facente riferimento al processo precedentemente lanciato e scrivere sul suo stream di input il goal relativo alla modalità di esecuzione scelta dall'utente, che è individuata facendo riferimento all'attributo "`run_project_mode`".

4.11 Esecuzione della gui SOCS-SI

Avendo l'esigenza di lanciare un'applicazione differente da SICStus Prolog, si è creato prima di tutto un nuovo tipo di configurazione di lancio. Per fare questo, si è aggiunto all'extension point

`org.eclipse.debug.core.launchConfigurationTypes` un elemento all'interno del quale si è provveduto a definire l'associazione del tipo di configurazione di lancio con la classe delegata al lancio vero e proprio dell'applicazione (in questo caso `LaunchSocsConfigurationDelegate`).

Analogamente al caso della compilazione di un progetto SCIFF, si è reso accessibile il lancio dell'applicazione SOCS-SI per mezzo di un pulsante sulla toolbar (Fig. 10). Quest'ultimo, anche in questo caso, permetterà l'accesso ad una finestra tramite la quale scegliere il progetto che configurerà il lancio del programma. La finestra in questione sarà, anche in questo caso, un'istanza della classe `ProjectDialog`, creata però settando il flag `socs_si` al valore `true`.

Al momento della pressione del tasto "ok" da parte dell'utente, avendo il flag sopra citato tale valore, verrà chiamato il metodo `startSOCS`. Questo creerà, se non già esistente, una configurazione di lancio del tipo `launchSocsConfigurationType` (differente quindi da quella utilizzata per il lancio di SCIFF), all'interno della quale sarà inserito il nome del progetto selezionato dall'utente.

A questo punto, la configurazione sarà passata alla classe `LaunchSocsConfigurationDelegate`. Questa, dapprima preleverà da essa il nome del progetto selezionato, poi si preoccuperà di estrapolare i file ics, sokb e history inseriti nel file project.pl relativo al progetto in esame. Fatto questo, li utilizzarà per creare la riga di comando che verrà poi utilizzata per il lancio del processo.

4.12 Help di SCIFF

Un altro importante elemento di un programma è l'help, il quale è presente anche in SCIFF. Il problema che ci si è posti è stato come integrarlo in maniera trasparente con Eclipse. Anche in questo caso si è adoperato uno strumento appositamente definito da un extension point identificato dalla stringa

`org.eclipse.help.toc` [8].

```
<toc label="Sciff Help"  >
   <topic label="User Manual" href="file:/D:/SCIFF/userman.html"/>
   <topic label="DeveloperManual" href="file:/D:/SCIFF/devman.html"/>
   <topic label="Sciff Home Page"
href="http://lia.deis.unibo.it/research/sciff/"/>
</toc>
```

Nell'ambito di quest'ultimo è stato possibile aggiungere un elemento denominato `toc` che rappresenterà la radice della struttura ad albero di documenti che vorremo aggiungere al sistema Help di Eclipse. Questo, poi, punterà ad un file xml che riporterà le indicazioni sulla struttura vera e propria che assumerà l'help relativo al nostro plugin.

Il file xml in questione, in questo caso, dovrà puntare ai file html presenti nella directory di installazione di SCIFF, per cui dovrà essere modificato dinamicamente. In particolare si è deciso di aggiornarne i percorsi dei file ogni volta che l'utente effettua modifiche alle preferenze tramite il metodo `Utility.refreshTocFile`. Questa funzione sostituirà ai prefissi dei nomi dei file .html presenti nell'attributo `href`, la stringa consistente nel path della directory di installazione di SCIFF.

4.13 Context help

In Eclipse viene messo a disposizione un meccanismo di Help context sensitive, ovvero che modifica i contenuti trasmessi all'utente a seconda del contesto nel quale ci si trova. Quando, ad esempio, ci si trova ad esplorare un menù, nel caso si incontri una voce di cui si voglia conoscerne le funzionalità, basterà posizionarsi sopra di essa con il mouse e premere il tasto F1 per far comparire una breve guida sul suo conto.

Per poter integrare questa funzionalità, si è proceduto all'espansione dell'extension point denominato `org.eclipse.help.contexts`, il quale offre la possibilità di specificare un file xml all'interno del quale sarà possibile inserire elementi denominati `context`, Questi associeranno l'identificativo di un contesto al contenuto che il context help fornirà all'utente.

Gli identificativi del contesto, invece, possono essere specificati programmaticamente oppure dichiarati nel file `plugin.xml` nell'ambito della definizione di un extension point che lo permetterà.

A questo proposito, l'obiettivo che ci si era proposti inizialmente, era quello di definire una breve guida ad ognuna delle action aggiunte da questo plugin alla toolbar. Ciò, però, non è stato possibile dal momento che nella definizione di queste nel file plugin.xml la voce `helpContextId` viene sfruttata solamente nel caso in cui l'action in considerazione sia parte di un menù. Si è così potuto optare per l'implementazione di questo tipo di funzionalità solamente nell'ambito dello “Run sciff” menù.

Quest'ultimo, definito dall'elemento `menu` interno ad `actionSet` nel contesto dell'extension point `org.eclipse.ui.actionSets`, è stato popolato con voci rappresentanti tutte le funzionalità messe a disposizione dei pulsanti presenti sulla toolbar, inserendovi quindi anche un riferimanto ad ognuna delle voci presenti nei menù a tendina delle action `start Sciff` e `run project`.

5 Conclusioni

Lo sviluppo di questo plugin ha portato ad ottenere un software che racchiude grosso modo tutte le funzionalità di base generalmente presenti all'interno di un IDE. Si è infatti provveduto a fornire all'utente una serie di elementi, quali evidenziazione della sintassi del linguaggio ed evidenziazione del bilanciamento delle parentesi, che possono rendere più confortevole la scrittura di applicazioni SCIFF.

Un altro obiettivo raggiunto, inoltre, è stato l'integrazione sotto lo stesso ambiente di diversi applicativi quali SICStus Prolog e la GUI SOCS-SI, rendendone più immediato l'utilizzo.

Possiamo inoltre confermare la bontà della scelta di Eclipse come applicativo per il quale progettare il plugin, infatti, grazie al meccanismo dei punti di estensione (extension point), come previsto è stato possibile suddividere lo sviluppo dell'intera applicazione in più moduli, semplificando la programmazione, rendendo così il tutto più estensibile.

Anche per quanto riguarda il problema dell'usabilità, citato tra gli obiettivi di questo progetto, si può essere soddisfatti del lavoro svolto. Il software ottenuto, infatti, assume un'interfaccia alquanto familiare per uno sviluppatore, riprendendo elementi tipici della maggior parte degli IDE, quali: finestra di navigazione delle risorse, editor, console, finestre di wizard e altri.

Guardando ai futuri sviluppi dell'applicazione, tuttavia, sono ancora molti gli elementi che potranno andare a completare le funzionalità del software. Durante l'utilizzo del programma, ad esempio, si potrà riscontrare l'esigenza di un output su console da parte di SICStus Prolog più razionalizzato e ordinato, filtrato, magari, di tutti i messaggi non necessari allo sviluppatore SCIFF. Altri punti di partenza per sviluppare il progetto, inoltre, potrebbero essere la possibilità di evidenziare i commenti multi-linea, oppure la possibilità da parte dell'IDE di attivare le icone di compilazione e lancio del progetto soltanto dopo che l'utente avrà fatto partire l'interprete SCIFF.

Bibliografia

[1] The SCIFF Abductive Proof Procedure, http://lia.deis.unibo.it/research/sciff/

[2] Eclipse Project , http://www.eclipse.org/

[3] SICStus Prolog, http://www.sics.se/isl/sicstuswww/site/index.html

[4] Eclipse Platform Technical Overview, http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html

[5] SOCS-SI, http://www.lia.deis.unibo.it/research/socs_si/socs_si.shtml

[6] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Compliance verification of agent interaction: a logic-based software tool. Applied Artificial Intelligence, 20(2-4):133-157, February-April 2006.

[7] Daum Berthold, Professional Eclipse 3 for Java Developers, Wiley, 2004

[8] Clayberg Eric, Rubel Dan, Eclipse Building Commercial-Quality Plug-Ins, Addison Wesley, 2006

Ringraziamenti

Al termine di questa importante tappa della mia vita, vorrei ringraziare coloro senza i quali questo non sarebbe stato possibile.

A mia madre e mio fratello, i quali hanno creduto in me, sempre, e che mi hanno riempito dell'affetto di cui avevo bisogno. A mio padre, per avermi donato il ricordo di un amore grande che mi ha sempre sostenuto. A mio zio Claudio, che avrebbe voluto molto assistere a questi momenti, e che sicuramente lo farà.

A mio nonno Mario e mia nonna Irene, custodi di valori ormai smarriti.

A tutti i parenti in generale .....che sono troppi e non ci starebbero (si dovranno accontentare)!

A tutti gli amici, quelli veri, che non hanno bisogno di leggersi in questo paragrafo, ma che si rispecchieranno tra le sue righe.