ESTENSIONE GUI TUPROLOG PER LA GESTIONE DELLE ECCEZIONE

Indice.

1 Introduzione.

Lo scopo di questo lavoro consiste nell'ottimizzare il programma tuProlog al fine di una corretta gestione dell'interazione tra utente e applicazione all'insorgere di eccezioni.

In particolare:

- Integrare il programma in modo che in modalità Console:
  - vengano catturate e mostrate dall'interfaccia le descrizioni delle eccezioni;
  - l'applicazione non termini nel caso si verifichi un'eccezione.
- Integrare il programma in modo che in modalità GUI:
  - vengano catturate e mostrate (in un tab opportuno) dall'interfaccia le descrizioni delle eccezioni
  - si disponga di un opzione per attivare o disattivare la ricezione delle eccezioni

2 Analisi Iniziale.

L'intervento da eseguire sul programma, nella sua attuale **versione 2.3.1 beta**, ha lo scopo principale di trasmettere alle interfacce attraverso cui l'utente interagisce un messaggio di errore nel momento in cui si verifica un'eccezione.

La versione attuale del motore gestisce già le eccezioni, per tanto è necessario determinare tre punti:

1. dove si catturano le eccezioni durante l'esecuzione di un predicato;
2. quali elementi devono mostrare l'errore;
3. come passare l'errore dal punto in cui si cattura all'elemento adibito alla visualizzazione.

Il motore dispone già di un meccanismo attraverso il quale è possibile notificare determinati componenti affinché visualizzino opportuni messaggi (solution, output, spy, warning) e sarà quindi possibile sfruttare un pattern simile per il mio scopo, in modo che si integri bene con le funzionalità già presenti.

Per rendere più modulare il lavoro lo ho suddiviso in tre parti:

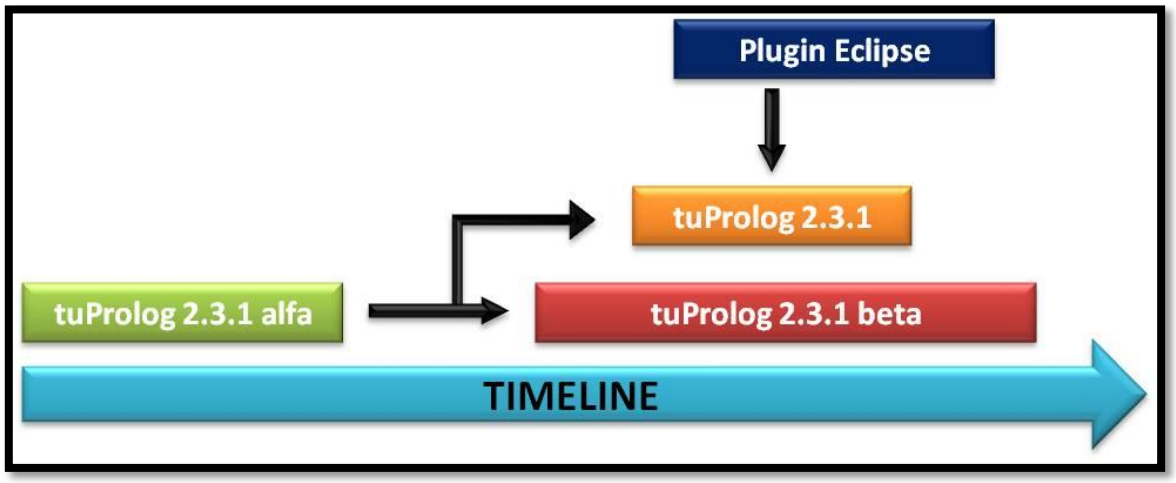
1. Modifiche al motore;
2. Modifiche alla console user interface;
3. Modifiche alla graphic user interface.

Durante lo svolgimento del lavoro è stato riscontrato un problema dovuto a riprogettazioni parallele del motore tuProlog.

Partendo dalla versione 2.3.1 alfa si è avviato un processo di re implementazione e risanamento dell'engine che è sfociato nella versione 2.3.1 beta. In parallelo si è anche sviluppato un progetto per la ristrutturazione del plugin tuProlog per Eclipse che, dovendosi appoggiare al motore tuProlog, ha assunto come base la versione 2.3.1 alfa.

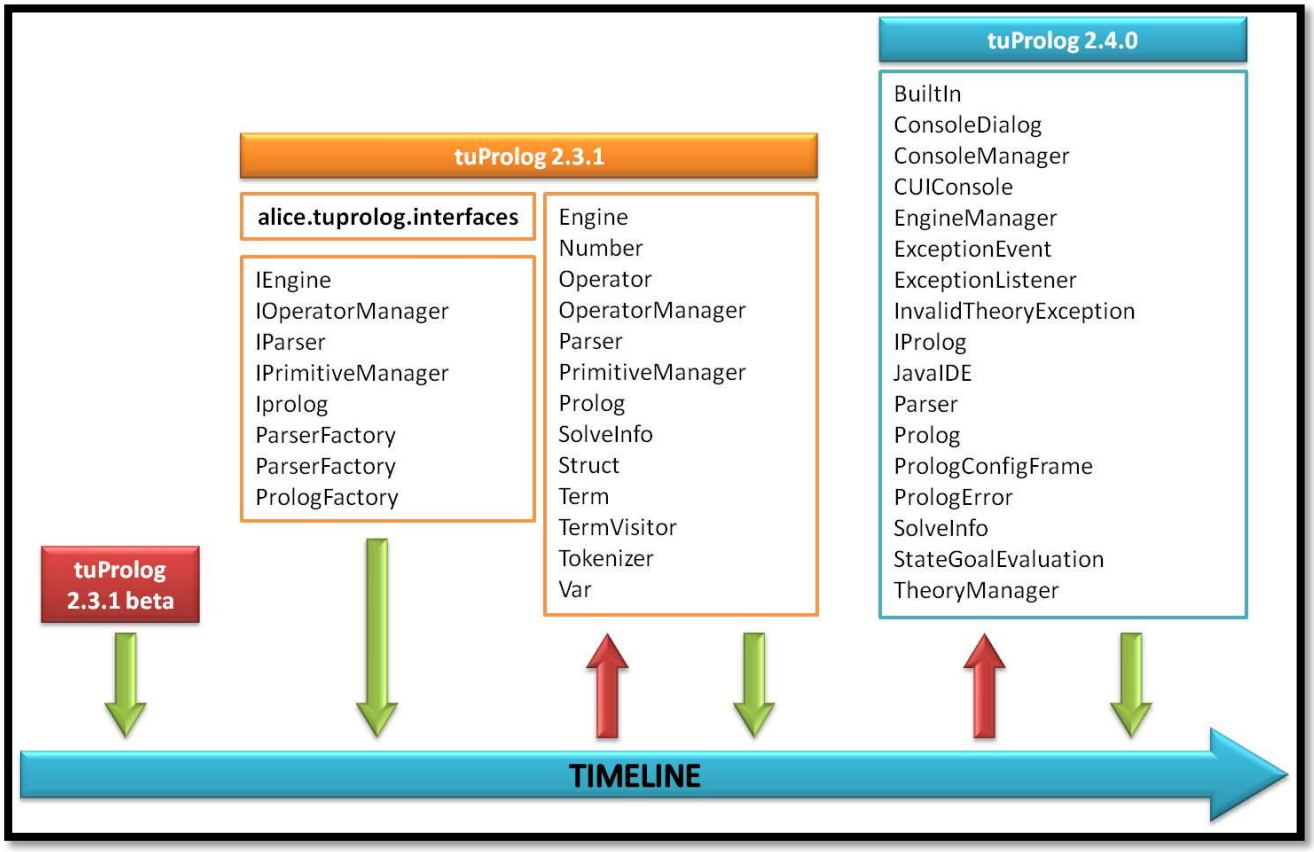
Entrambi i percorsi hanno agito sul motore portando sia alla risoluzione di alcuni bug, sia alla introduzione di elementi come i "generics" (nel primo progetto) e di interfacce di comunicazione con il plugin (nel secondo progetto).

La situazione a cui si andava incontro era dunque la seguente:



Per poter estendere l'interazione utente-applicazione al supporto di una corretta comunicazione delle eccezioni era quindi necessario individuare prima su quale versione del motore dover agire. Considerando il fatto che le migliorie apportate da entrambe le versioni prodotto sarebbero dovute essere tutte presenti in una futura release del progetto si è optato per una loro fusione.

La versione di base da cui si partirà sarà la 2.3.1 beta. Questa sarà aggiornata delle modifiche introdotte nella versione 2.3.1. Infine saranno apportate le integrazioni necessario al supporto della gestione delle eccezioni.



3 Aggiornamento del motore di base

L'aggiornamento del motore di base (versione 2.3.1 beta) alla versione re implementata per il plugin ha comportato due operazioni:

1. introduzione di un nuovo package "`alice.tuprolog.interfaces`" contente interfacce volte ad una maggiore separazione del motore dal plugin (in questo modo il plugin può funzionare anche utilizzando diverse versioni di motori prolog);
2. valutazione delle differenze introdotte nella versione del motore del nuovo plugin e loro inserimento nelle corrispondenti classi del motore di base.

3.1 Package `alice.tuprolog.interfaces`

Dato che nel progetto di base non era presente il package "`alice.tuprolog.interfaces`" e le relative classi contenute, sono state semplicemente aggiunte le seguenti interfacce:

- IEngine
- IOperatorManager
- IParser
- IPrimitiveManager
- IProlog
- ParserFactory
- ParserFactory
- PrologFactory

3.2 Classe `Engine`

Engine implementa ora l'interfaccia `IEngine`.

3.3 Classe `Number`

È stato aggiunto il metodo "`public void accept(TermVisitor tv)`".

3.4 Classe `Operator`

È stato reso "`public`" il metodo "`Operator(String name_,String type_,int prio_)`".

3.5 Classe `OperatorManager`

È stata resa "**public**" la classe.
OperatorManager implementa ora l'interfaccia "IOperatorManager" ed il metodo "**public** `IOperatorManager clone()`".

La classe annidata "`OperatorRegister`" implementa ora l'interfaccia "`Cloneable`" ed il metodo "**public** `Object clone()`" (per il quale è stato necessario importare il package "`java.util.Iterator`".

3.7 Classe `Parser`

`Parser` implementa ora l'interfaccia "`IParser`".

Sono stati aggiunti i campi

- **private** `HashMap<Term, Integer> offsetsMap;`
- **private int** `tokenStart`.

Sono stati aggiunti i costruttori:

- **public** `Parser(OperatorManager op, String theoryText, HashMap<Term, Integer> mapping);`
- **public** `Parser(String theoryText, HashMap<Term, Integer> mapping).`

È stato modificato il costruttore:

- **public** `Parser(String theoryText).`

Sono stati modificati I seguenti metodi:

- **private** `IdentifiedTerm exprA(int maxPriority, boolean commaIsEndMarker);`
- **private** `IdentifiedTerm exprB(int maxPriority, boolean commaIsEndMarker);`
- **private** `IdentifiedTerm parseLeftSide(boolean commaIsEndMarker, int maxPriority);`
- **private** `Term expr0();`
- **private** `LinkedList<Term> expr0_arglist().`

Sono stati aggiunti i seguenti metodi:

- **private** `IdentifiedTerm identifyTerm(int priority, Term term, int offset);`
- **private void** `map(Term term, int offset);`
- **public** `HashMap<Term, Integer> getTextMapping();`
- **public int** `getCurrentOffset();`
- **public int**`[] offsetToRowColumn(int offset).`

3.8 Classe `PrimitiveManager`

`PrimitiveManager` implementa ora `IPrimitiveManager` e il metodo "**public boolean** `containsTerm(String name, int nArgs)`".

3.9 Classe `Prolog`

`Prolog` implementa ora `IProlog`.

3.10 Classe `Struct`

È stato aggiunto il metodo "`public void accept(TermVisitor tv)`".

3.11 Classe `Term`

È stata aggiunta la dichiarazione "`public abstract void accept(TermVisitor tv)`".

3.12 Classe `TermVisitor`

È stata aggiunta la classe `TermVisitor`.

3.13 Classe `Tokenizer`

È stata resa "`public`" la classe.
Sono stati aggiunti i seguenti campi:

- `private int tokenOffset;`
- `private int tokenStart;`
- `private int tokenLength;`
- `private String text = null.`

È stato modificato il costruttore "`public Tokenizer(String text)`".
È stato reso pubblico il metodo "`Token readToken()`".
Sono stati modificati i seguenti metodi:

- `Token readNextToken() throws IOException, InvalidTermException;`

Sono stati aggiunti I seguenti metodi:

```
public int lineno();
```

- `public int tokenOffset();`
- `public int tokenStart();`
- `public int[] offsetToRowColumn(int offset);`
- `String removeTrailing(String input,int tokenOffset);`
- `private int tokenConsume() throws IOException;`
- `private void tokenPushBack().`

3.14 Classe `Var`

È stato aggiunto il metodo "`public void accept(TermVisitor tv)`".

3.15 Classe `SolveInfo`

È stato modoficatp il metodo "`public String toString()`" per mostrare il risultato "`halt.`" In caso di eccezione.

4 Gestione delle eccezioni: motore prolog

Di seguito elenco le classi che subiranno modifiche o che saranno create per permettere la comunicazione e la presentazione delle eccezioni:

- `StateGoalEvaluation` **package** `alice.tuprolog`
- `EngineManager` **package** `alice.tuprolog`
- `Prolog` **package** `alice.tuprolog`
- `ExceptionEvent` **package** `alice.tuprolog.event`
- `ExceptionListener` **package** `alice.tuprolog.event`
- `PrologError` **package** `alice.tuprolog`

Durante il test dell'applicazione è stato rilevato un comportamento anomalo nell'esecuzione del predicato "`halt.`". Per standardizzare il comportamento del motore si sono modificati i metodi interessati nella seguente classe:

- `BuiltIn` **package** `alice.tuprolog`

Su segnalazione di un docente estero è stato verificato un bug nel caricamento di una teoria con errori di sintassi. Il relativo PrologError generato indica sempre che l'errore si trova alla linea -1 e posizione -1 della teoria. Il problema, non presente fino alla versione 2.0 (nella 2.1 la sintassi veniva accettata senza rilanciare errori), è una conseguenza della reingegnerizzazione del `TheoryManager` e del `Parser` avvenuta tra la versione 2.0.1 e la 2.1.0. Con l'aggiornamento del motore il problema è stato risolto e la soluzione adottata ha permesso di apportate le necessarie modifiche per supportare la creazione di errori di sintassi in modo corretto.

Le classi che subiranno modifiche sono le seguenti:

- `InvalidTheoryException` **package** `alice.tuprolog`
- `TheoryManager` **package** `alice.tuprolog`
- `Parser` **package** `alice.tuprolog`
- `InvalidTermException` **package** `alice.tuprolog`
- `BasicLibrary` **package** `alice.tuprolog`
- `IOLibrary` **package** `alice.tuprolog`

4.1 Classe `StateGoalEvaluation`

`StateGoalEvaluation` permette la valutazione di un goal attraverso il metodo "**void** `doJob(Engine e)`" che richiama delle libreria una funzione opportuna. Questo avviene all'interno di un blocco **try** seguito da due blocchi **catch**. Il primo blocco catch cattura eccezioni di tipo `HaltException` (ossia non gestibili) mentre il secondo cattura eccezioni di tipo `Throwable` (che possono essere istanze di `PrologErrore` oppure `JavaException`).

È necessario modificare il secondo blocco catch del metodo "**void** `doJob(Engine e)`" per ricavare dall'eccezione un messaggio descrittivo chiaro e passarlo all'istanza `Prolog`. Per fare ciò occorre passare

attraverso una classe intermedia. Sfruttando il riferimento all'istanza `Engine` (passata come parametro nel metodo) si può accedere all'istanza `EngineManager` che si occuperà di notificare l'istanza `Prolog`.

4.2 Classe `EngineManager`

`EngineManager` si occupa già di notificare l'istanza `Prolog` per messaggi di spy e di warning. È sufficiente, quindi, aggiungere il metodo "`void exception(String message)`" per passare un messaggio relativo ad un'eccezione.

4.3 Classe `Prolog`

`Prolog` gestisce diversi tipi di ascoltatori al suo interno e per ciascuna tipologia definisce:

1. un `ArrayList` contenitore;
2. una serie di metodi accessori per aggiungere, rimuovere o restituire gli ascoltatori;
3. un campo di stato, che espone metodo di set e di get, che discrimina se `Prolog` debba notificare gli ascoltatori;
4. un metodo per indicare dall'esterno di notificare un messaggio agli ascoltatori;
5. un metodo per notificare tutti gli ascoltatori.

Gli ascoltatori di eventi di tipo `ExceptionListener` si sono gestiti al pari degli altri e quindi sono state apportate le seguenti aggiunte (rispettando i punti definiti prima):

1. `private ArrayList<ExceptionListener> exceptionListeners`
2. `public synchronized void addExceptionListener(ExceptionListener l)`
   `public synchronized void removeExceptionListener(ExceptionListener l)`
   `public synchronized void removeAllExceptionListeners()`
   `public synchronized List<ExceptionListener> getExceptionListenerList()`
3. `private boolean exception`
   `public synchronized void setException(boolean state)`
   `public synchronized boolean isException()`
4. `public void exception(String m)`
5. `protected void notifyException(ExceptionEvent e)`

4.4 Classe `ExceptionEvent`

`ExceptionEvent` (`package alice.tuprolog.event`) estende `EventObject` e espone un campo `String` per contenere il messaggio d'errore.

4.5 Classe `ExceptionListener`

`ExceptionListener` estende `EventListener` e espone il metodo "`public abstract void onException(ExceptionEvent e)`".

4.6 Classe `PrologError`

`PrologError` estende `Throwable` ed espone metodi statici per la creazione di istanze di `PrologError` rappresentanti determinate tipi di errore. La presentazione di un errore in termini di stringa descrittiva non è adatta siccome contiene ridondanze e non è user friendly. Ho aggiunto, quindi, un campo "`private String descriptionError`" adibito ad essere inizializzato in un nuovo costruttore "`public PrologError(Term error, String descriptionError)`".

Il nuovo costruttore effettua le stesse operazioni di quello originale per non perdere compatibilità. Il costruttore originale non è stato rimosso siccome è invocato dal metodo "`public boolean throw_1(Term error) throws PrologError`" della classe `BasicLibrary`.

Ciascun metodo statico dedicato alla creazione di un PrologError è stato modificato affinché generasse una stringa descrittiva dell'errore da passare al nuovo costruttore creato. I metodi modificati sono i seguenti:

1. `public static PrologError instantiation_error(EngineManager e, int argNo)`
2. `public static PrologError type_error(EngineManager e, int argNo, String validType, Term culprit)`
3. `public static PrologError domain_error(EngineManager e, int argNo, String validDomain, Term culprit)`
4. `public static PrologError existence_error(EngineManager e, int argNo, String objectType, Term culprit, Term message)`
5. `public static PrologError permission_error(EngineManager e, String operation, String objectType, Term culprit, Term message)`
6. `public static PrologError representation_error(EngineManager e, int argNo, String flag)`
7. `public static PrologError evaluation_error(EngineManager e, int argNo, String error)`
8. `public static PrologError resource_error(EngineManager e, Term resource)`
9. `public static PrologError syntax_error(EngineManager e, int line, int position, Term message)`
10. `public static PrologError system_error(Term message)`

4.7 Classe `BuiltIn`

`BuiltIn` è una libreria di predicati. I seguenti metodi sono stati commentati e implementati in modo diverso:

1. `public boolean halt_0() throws HaltException`
2. `public boolean halt_1(Term arg0) throws HaltException, PrologError`

Il primo metodo rilanciava una `HaltException`. Il secondo metodo analizzava il `Term` passato come argomento e se non presentava errori di sintassi o di tipo (che avrebbe altrimenti rilanciato come `PrologError`) rilanciava una `HaltException`.

Attualmente i due metodi non rilanciano `HaltException` e terminano l'applicazione (comportamento standard).

4.8 Classe `InvalidTheoryException`

InvalidTheoryException rappresenta un'eccezione dovuta al fatto che si è specificata una teoria non valida. Presentava due attributi "`line`" e "`pos`" inizializzati a `-1` e due costruttori:

```
public InvalidTheoryException(int line, int pos)
```

```
public InvalidTheoryException(String message)
```

Il primo non veniva mai invocato da alcuna classe, pertanto i due attributi non venivano aggiornati e la costruzione di un `PrologError` per un errore di sintassi riceveva come parametri i due valori impostati a `-1`. È stato aggiunto un ulteriore costruttore "`public InvalidTheoryException(String message, int clauseNumber)`" in cui si sfrutta un'informazione recuperabile attraverso il `TheoryManager` che indichi almeno il numero della clausola in cui si è riscontrato il primo errore di sintassi.

4.9 Classe `TheoryManager`

`TheoryManager` gestisce le calusole e le torie che costituiscono il database di tuProlog. Nel metodo "`void consult(Theory theory, boolean dynamicTheory, String libName) throws InvalidTheoryException`" si consulta la teoria speficiata dall'utente. L'algoritmo itera ogni clausola della toeria e se viene lanciata un viene catturata come `InvalidTermException` e rilanciata come `InvalidTheoryException`.

Attualmente è stato inserito un parametro che tiene conto delle iterazione (ad ogni iterazione viene consultata una clausola della teoria) in modo che nel momento in cui si verifica un'eccezione si possibile ricavare la clausola in errore e sfruttare il nuovo costruttore di `InvalidTheoryException` per aggiungerla alle informazioni di errore.

4.10 Classe `Parser`

Nella classe `Parser`, ad ogni metodo che rilancia una `InvalidTermException`, è stato modificato il costruttore con il quale si creava un'istanza di questa classe errore. Il nuovo costruttore permette di indicare, oltre al messaggio di errore, il numero di linea e la posizione (attraverso il metodo "`public int[] offsetToRowColumn(int offset)`" della classe `Tokenizer`) in cui si è verificato l'errore.

4.11 Classe `InvalidTermException`

Nella classe `InvalidTermException` sono stati aggiunti due campi ("`public int line = -1`" e "`public int pos = -1`") per la memorizzazione della linea e della posizione in cui si è verificato un errore. È stato inoltre aggiunto il costruttore "`public InvalidTermException(String message, int line, int pos)`" per inizializzare i due campi oltre che passare il messaggio di errore.

4.12 Classe `BasicLibrary`

Nella classe `BasicLibrary` è stata cambiata l'invocazione del metodo di `PrologError` per la creazione di un errore di sintassi nei seguenti metodi:

- `public boolean set_theory_1(Term th);`
- `public boolean add_theory_1(Term th).`

4.13 Classe `IOLibrary`

Nella classe `IOLibrary` è stata cambiata l'invocazione del metodo di `PrologError` per la creazione di un errore di sintassi nel metodo "`public boolean read_1(Term arg0)`".

4.14 Classe `IProlog`

IProlog fa parte delle interfacce aggiunte durante l'aggiornamento del motore allo scopo di astrarre l'uso dell'engine dal plugin. Dato che il motore dispone ora della gestione delle eccezioni sono state aggiunti all'interfaccia i metodi relativi alla gestione degli ascoltatori di eventi di eccezioni:

- `void addExceptionListener(ExceptionListener l);`
- `void removeExceptionListener(ExceptionListener l);`
- `void removeAllExceptionListeners().`

5 Gestione delle eccezioni: console user interface

La classe adibita alla comunicazione tra utente e motore è la seguente:

- `CUIConsole` `package alice.tuprologx.ide`

5.1 Classe `CUIConsole`

`CUIConsole` è già strutturata in modo tale da essere sensibile a particolare tipi di eventi. Per questo si registra tra gli ascoltatori dell'istanza Prolog per essere notificata in caso di `OutputEvent`, `SpyEvent` e `WarningEvent`.

Per essere sensibile agli eventi di tipo `ExceptionEvent` la `CUIConsole` implementerà l'interfaccia `ExceptionListener`, il relativo metodo "`public void onException(ExceptionEvent e)`" (che produrrà una stampa a video del eccezione) e si registrerà fra gli ascoltatori dell'istanza `Prolog`.

La condizione di terminazione dell'applicazione in seguito ad una eccezione scaturiva da un controllo effettuato nel metodo "`void solveGoal(String goal)`" sullo stato di HALT del motore. Dato che il verificarsi di un'eccezione non gestita pone il motore nello stato HALT il controllo causava inesorabilmente la terminazione ed è stato, quindi, commentato.

Il successivo testing di un predicato che scatenasse un'eccezione ha portato alle seguenti conclusioni:

1. la presentazione delle eccezioni mostra una corretta stampa del messaggio di errore;
2. la presentazione della soluzione non è adeguata in quanto si otteneva "`no.`" al pari di un qualunque altro predicato che fallisse.

5.2 Errore presentazione soluzione

Il problema riscontrato nella seconda conclusione è una conseguenza dovuta all'eliminazione del controllo sullo stato di `HALT`. Siccome si possono avere quattro precisi stati finali (definiti dalla classe `EngineManager`) `HALT = -1`, `FALSE = 0`, `TRUE = 1` e `TRUE_CP = 2` se il programma superava il controllo originale sullo stato di HALT era naturale restituire un risultato che fosse o "`si.`" o "`no.`".

Dato che attualmente lo stato di `HALT` non produce la terminazione è stato inserito un controllo sullo stato del motore nel caso la soluzione non abbia avuto successo. Se ci si trova nello stato di `HALT` verrà stampato a video "`halt.`" altrimenti "`no.`".

5.3 Bug esecuzione predicati

Si è riscontrato un bug, non presente nelle versioni precedenti di tuProlog, che comportava la terminazione dell'applicazione a causa di una `StringIndexOutOfBoundsException` scatenata nel metodo "`private String solveInfoToString(SolveInfo result)`" ove era stato omesso un controllo.

6 Gestione delle eccezioni: graphic user interface

Di seguito elenco le classi che subiranno modifiche per permettere la comunicazione e la presentazione delle eccezioni:

- `JavaIDE` **`package`** `alice.tuprologx.ide`
- `ConsoleManager` **`package`** `alice.tuprologx.ide`
- `PrologConfigFrame` **`package`** `alice.tuprologx.ide`
- `ConsoleDialog` **`package`** `alice.tuprologx.ide`

6.1 Classe `JavaIDE`

`JavaIDE` inizializza i componenti grafici e le dipendenze fra di loro. Il componente grafico adibito alla ricezione e presentazione delle eccezioni è implementato dalla classe `ConsoleDialog`. Quest'ultimo verrà quindi registrato anche fra gli ascoltatori di `ExceptionEvent`.

`JavaIDE` inizializza anche il componente che gestisce le opzioni che l'utente può impostare (`PrologConfigFrame`). Si preoccuperà quindi, rispettando lo stato del motore (l'istanza `Prolog`) nei casi in cui sia sensibile o meno alle eccezioni (di default lo è), di inizializzare i parametri relativi alla configurazione delle eccezioni. Inoltre registra. Dato che motore e interfaccia devono essere avvisati nel caso in cui l'utente modifichi le opzioni riguardanti le eccezioni, `JavaIDE` registra nel `PrologConfigFrame` il `ConsoleManager` e la `ConsoleDialog`.

6.2 Classe `PrologConfigFrame`

`PrologConfigFrame` espone all'utente un'interfaccia mediante la quale settare dei parametri. L'interfaccia organizzava i componenti grafici attraverso un `GridBagLayout`. Questa scelta non risulta abbastanza elastica per l'aggiunta di nuove opzioni e oltretutto non è visivamente adatta siccome da un senso di disordine.

Le modifiche seguenti, ottenute anche grazie al supporto del plugin per Eclipse denominato Jigloo, sono state apportate nel metodo "**`private void`** `initComponents()`". Il nuovo frame è ora organizzato con un `BorderLayout` che ospita tre pannelli nelle seguenti posizioni:

- nord: visualizza le opzioni di display;
- centro: visualizza le opzioni del motore;
- sud: visualizza i bottoni per accettare accettare le impostazioni o annullarle.

Questi pannelli, di cui i primi due circondati da un bordo con etichetta relativa al tipo di opzioni che gestiscono, presentano un `BoxLayout` con allineamento verticale. In questo modo si fornisce un'efficiente struttura per l'aggiunta delle opzioni attuali e di ulteriori impostazioni future.

Per quanto riguarda la configurazione delle eccezioni si è fornita la possibilità di attivarne o disattivarne la visualizzazione. A questo scopo dovranno essere informati sia il `Prolog` (che si occupa di notificare gli

ascoltatori di eventi di tipo `ExceptionEvent`) sia la `ConsoleDialog` (che dovrà mostrare all'utente se stia ricevendo o meno le eccezioni).

Per gestire i le opzioni è stato usato un'oggetto `PropertyChangeSupport`. Questo permette di aggiungere e aggiornare proprietà passando il nome e il valore della proprietà. Permette inoltre di registrare degli ascoltatori in modo tale che vengano notificati nel momento in cui si verifichi un evento `PropertyChangeEvent` in seguito all'aggiunto o all'aggiornamento di una proprietà.

Come detto nel precedentemente, `JavaIDE` registrerà fra gli ascoltatori di `PropertyChangeSupport` il `ConsoleManager` e la `ConsoleDialog`.

Nel metodo **`public void`** `reload()`, invocato quando la finestra viene riaperta, è stato aggiunto il settaggio del `JCheckBox` che indica se attualmente il motore sta notificando alle interfacce le eccezioni.

Nel metodo "**`public void`** `ok()`", invocato quando vengono accettate le impostazioni, è stato aggiunto il settaggio della configurazione delle eccezioni nell'oggetto `PropertyChangeSupport` (che avviene attraverso il nuovo metodo definito "**`public void`** `setNotifyExceptionEvent(`**`boolean`** `newValue)`"). Questo comporta la notifica di un `PropertyChangeEvent` a tutti i `PropertyChangeListener`.

6.3 Classe `ConsoleManager`

L'istanza `Prolog` deve essere sensibile alle variazioni di configurazione relative alle eccezioni. Per propagare le modifiche che l'utente apporta dal set di opzioni configurabili al motore si sfrutta la classe `ConsoleManager`.

`ConsoleManager` implementa `PropertyChangeListener` e il relativo metodo "**`public void`** `propertyChange(PropertyChangeEvent event)`". All'interno di questo metodo (che gestisce già la modifica relativa al tempo limite di esecuzione del motore oltre il quale di ferma l'esecuzione) è stato aggiunto un controllo. Nel caso la proprietà cambiata riguardi le eccezioni si notifica opportunamente `Prolog`.

6.4 Classe `ConsoleDialog`

ConsoleDialog è il componente grafico adibito a mostrare i risultati. I diversi tipi di eventi da visualizzare sono organizzati in tab attraverso un oggetto `JTabbedPane` di cui si mantiene una costante statica per identificarli.

`ConsoleDialog` dovrà essere sensibile sia agli eventi di tipo `ExceptionEvent` (implementerà l'interfaccia `ExceptionListener`) sia alle modifiche operate attraverso il `PrologConfigFrame` (implementa già l'interfaccia `PropertyChangeListener`).

Il nuovo tab dedicato alle eccezioni dovrà uniformarsi al comportamento di quelli già presenti ed essere essere integrato di una funzionalità che mostri se le eccezioni sono attive (in termini di notifica) sul motore.

A questo scopo sono state apportate le seguenti modifiche:

- creata una variabile statica che identifichi il nuovo tab (“`private static final int EXCEPTION_INDEX = 4`”);
- creato un `JTextPane` (“`private JTextPane exception`”) nel quale scrivere i messaggi di errore e definiti due stili (uno in corsivo e uno no) con i quali scrivere all’interno del componente;
- creata una variabile boolean (“`private boolean exceptionEnabled;`”) che rispecchi lo stato del motore rispetto alla notifica delle eccezioni;
- aggiunto al metodo “`protected void clear()`”, dedicato a cancellare il contenuto del tab selezionto, il controllo per pulire il tab delle eccezioni;
- aggiunto il metodo “`public void onException(ExceptionEvent event)`” attraverso il quale il tab eccezioni presenterà l’eccezione ricevuta e inoltre l’etichetta del tab assumerà un coloro rosso chiaro;
- aggiunto al metodo “`public void propertyChange(PropertyChangeEvent event)`”, che si occupa di ricevere gli eventi dovuti a modifiche nelle impostazione del `PrologConfigFrame`, un controllo sulla proprietà della notifica delle eccezioni;
- aggiunto un controllo al metodo “`public void stateChanged(ChangeEvent arg0)`” che si occupa di abilitare il bottone che richiama il metodo “`protected void clear()`” nel caso il tab visualizzato dall’utente non sia vuoto per settarlo anche in quello delle eccezioni;
- aggiunto il metodo “`public void setExceptionEnabled(boolean enable)`” attraverso il quale si setta nella classe il parametro che riflette lo stato del motore riguardo alla notifica delle eccezioni;
- aggiunto il metodo “`private void setExceptionJTextPaneRendering()`” che aggiorna la view del tab a seconda che si siano abilitate o no le eccezioni. Nel caso si siano disabilitate l’etichetta del tab e lo sfondo del JTextPane manterranno un colore grigio chiare e verrà scritto all’interno del pannello “`Exception notification disabled`” in corsivo.

Come nella `CUIConsole` anche qui una soluzione in stato di `HALT` veniva trattata come una soluzione in stato `FALSE` mostrando un “`no.`”. Attualmente è stato aggiunto un controllo nel metodo “`private void showSolution(SolveInfo info)`” in modo che in caso di eccezione non gestita sia presentato il risultato “`halt.`”.

6.5 Locazione delle costanti di versione

Vista la necessità di aggiornare la versione del programma si sono individuate le locazione delle due costanti rappresentanti la versione del motore e dell’IDE di tuProlog.

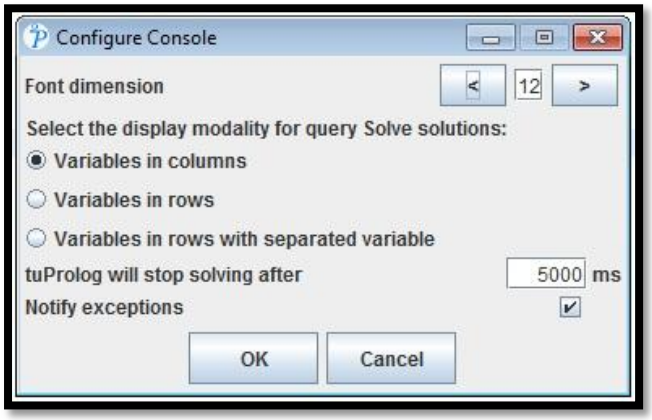
La prima risiede nella classe `Prolog` (`package alice.tuprolog`) e consiste della costante “`private static final String VERSION = "2.4.0 beta"`”.

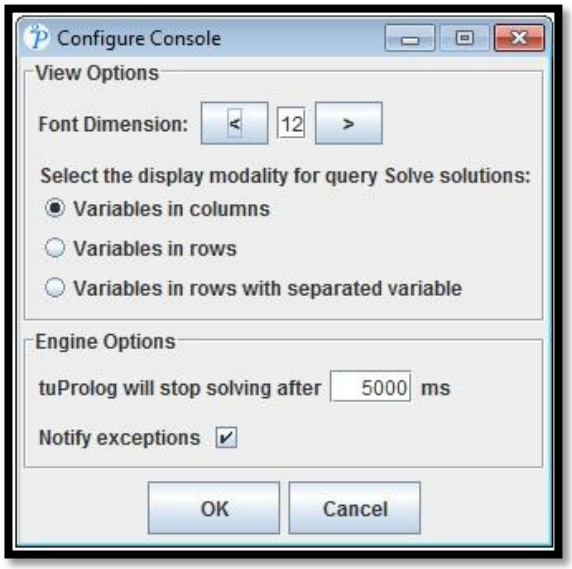
La seconda si trova nella classe `AboutFrame` (`package alice.tuprologx.ide`) e consiste di una JLabel dichiarata nel metodo “`private void initComponents()`”.

7 Screenshot

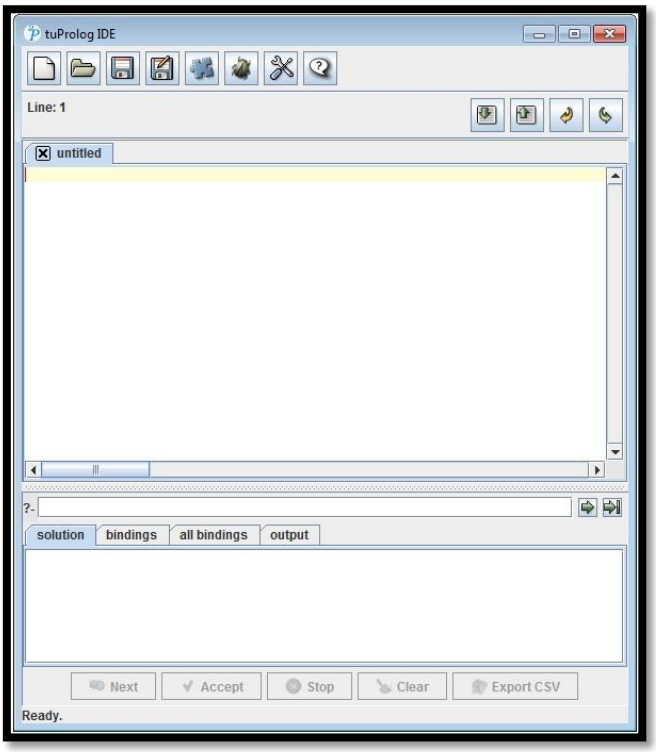
Gli screenshot successivi danno un'idea del risultato finale del lavoro eseguito per quanto riguarda l'interazione utente-programma.

7.1 Configure Console

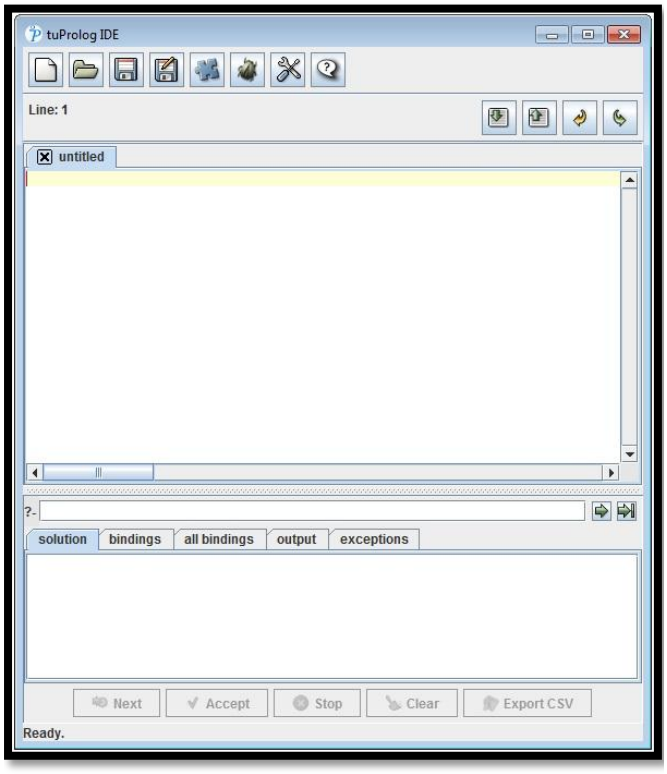




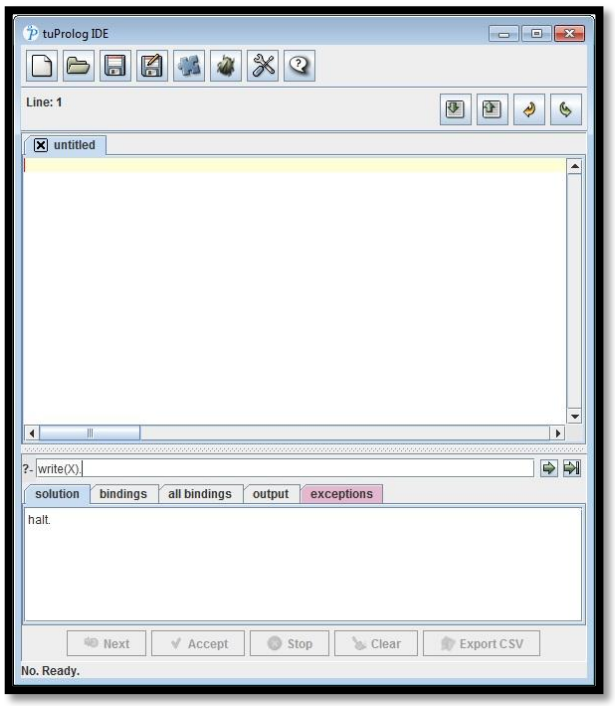
7.2 Graphic user interface



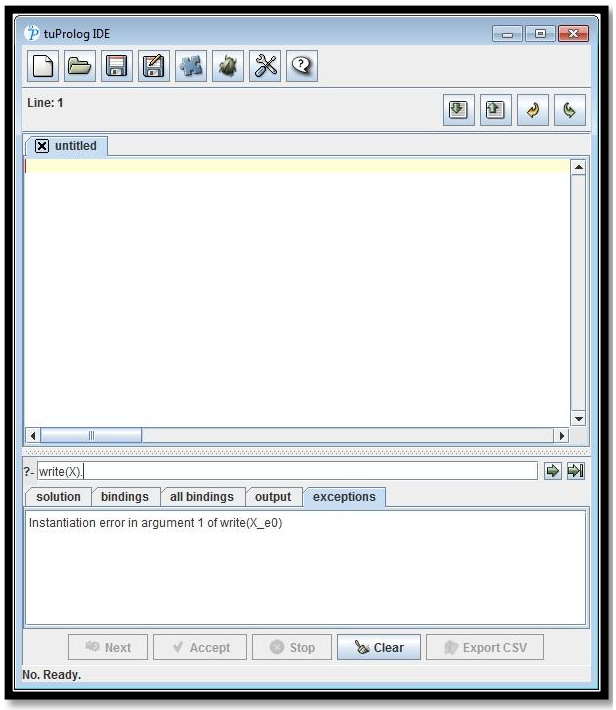
3 Vecchia GUI



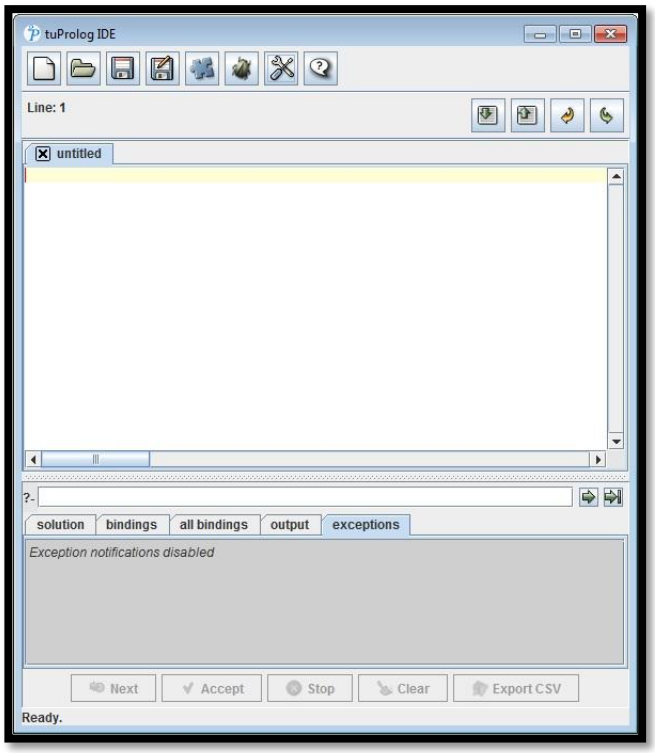
2 Nuova GUI



3 Nuova GUI



4 Nuova GUI



5 Nuova GUI

8 Plugin per Eclipse di tuProlog

L'estensione della gestione delle eccezioni aggiunta alla versione stand-alone di tuProlog risulta utile anche al plugin per Eclipse. La versione 2.3.1. beta, grazie all'aggiornamento, dispone ora delle necessarie interfacce per consentire la comunicazione con il nuovo plugin (sviluppato utilizzando la versione 2.3.1 ottenuta dalla versione 2.3.1 alfa).

Il progetto che costituisce il plugin si appoggia su di un motore prolog referenziandolo come libreria esterna. Pertanto occorrerà semplicemente sostituirlo con l'ultima versione del motore (2.4.0) e settare opportunamente le informazioni di buildpath.

Si sono riscontrati alcuni problemi nel file MANIFEST.MF siccome nella versione 2.4.0 si sono eliminati alcuni package. È stato sufficiente rimuovere questi dalla lista dei package del motore.

Infine sarà necessario aggiungere al plugin la possibilità di porre in ascolto di eventi di tipo eccezione le classi necessarie al fine di poter far giungere all'interfaccia grafica un messaggio di errore.

Dato che supporta già un meccanismo per mostrare i messaggi provenienti dal motore (result, output, spy, warning) è sufficiente riprodurre la modalità di comunicazione già adottata.

8.1 Plugin tuProlog: sostituzione motore 2.4.0

Dato che il plugin era statao sviluppato utilizzando il motore 2.3.1, appositamente modificato, è stato necessario sostituire questo con il nuovo motore 2.4.0. Grazie all'aggiornamento che ha aggiunto le interfacce di comunicazione è risultata immediata la risoluzione dei problemi di buildpath relativi ai metodi che il plugin necessitava dalla libreria corrispondente al motore. Sono state poi apportate alcune modifiche relative alla configurazione del plugin.

Operazioni:

- rimosso motore "2p.2.3.1.jar";
- introdotto e aggiunto al buildpath motore "2p2.4.0.jar";
- "MANIFEST.MF" -> "Runtime" -> "Exported Packages" rimossi tutti e riaggiunti (alcuni non sono più presenti);
- "MANIFEST.MF" -> "Runtime" -> "Classpath" rimossa libreria "2p2.3.1.jar" e aggiunto "2p2.4.0.jar";
- "MANIFEST.MF" -> "Vuild-properties" controllo riferimenti nuova libreria motore.

8.2 Classe `ConsoleView`

`ConsoleView` dispone ora di un ulteriore CTabItem dedicato alle eccezioni definito e costruito all'interno del metodo "`public void createPartControl(Composite parent)`". Nel metodo "`public void setQuery( PrologQuery query )`" è stata aggiunta l'istruzione per pulire l'area di testo dedicata alle eccezioni quando si setta una nuova query. Nel metodo "`private void refreshResultViewer()`" è stata aggiunta l'istruzione per rinfrescare l'area in seguito all'esecuzione di una query che, in caso di eccezione, visualizzerà il messaggio di errore.

8.3 Classe `PrologQueryResult`

`PrologQueryResult` fornisce le informazioni riguardanti lo stato finale di una query come il risultato, l'output e i messaggi di debug (spy e warning). È stato quindi aggiunto un campo per contenere i messaggi di eccezione e i necessari metodi per ottenerli e settarli "`public String getException()`" e "`public void setException(String exception)`".

8.4 Classe `EventListener`

EventListener implementa `SpyListener` e `OutputListener` e riceve quindi gli eventi relativi provenienti dal motore. È stato quindi inserita l'ulteriore implementazione dell'interfaccia `ExceptionListener`, poi implementato il relativo metodo "`public void onException(ExceptionEvent arg0)`" e quindi fornito il supporto per la memorizzazione del messaggio di eccezione oltre che i necessari metodi accessori "`public String getException()`" e "`public void setException(String exception)`".

8.5 Classe `PrologEngine`

PrologEngine dispone ora dei due seguenti metodi per la gestione degli ascoltatori di eventi di tipo ExceptionEvent: "`public void addExceptionListener(ExceptionListener exceptionListener)`" e "`public void removeExceptionListeners()`".

Nel metodo "`public String next()`" è stato cambiato il risultato fornito in caso di eccezione. Invece di visualizzare "`no.`" Se il risultato si trova in stato di `HALT` viene mostrato "`halt.`".

8.6 Classe `PrologQueryFactory`

Nel metodo "`public boolean executeQueryWS(PrologQuery query)`" è stato inserita l'istruzione per registrare, all'esecuzione di una query, l'`EventListener` fra gli ascoltatori di eventi di tipo `ExceptionEvent` dell'engine. Di conseguenza è stato anche settato nel `PrologQueryResult` l'eventuale messaggio d'eccezione, relativo allo stato finale dell'esecuzione della query, recuperato dall'`EventListener` stesso.

8.7 Guida per l'esportazione del plugin tuProlog per Eclipse

Di seguito sonor iportati i passi da seguire per l'esportazione del plugin:

1. cliccare sul progetto alice.tuprologx.eclipse con il tasto destro e selezionare "Export..";
2. selezionare "Deployable plugins and fragments" sotto la voce "Plug-in Development";
3. nella finestra con l'elenco dei frammenti selezionare "it.alice.unibo.tuprologx.eclipse";
4. sotto il pannello "Destination" selezionare dove salvare il JAR prodotto;
5. sotto il pannello "Options" selezionare:
    - Export source: "include source in exported plug-ins"
    - Package plug-ins as individual JAR archives
    - allow for binary cycles in target platform

Una volta ottenuto il plugin con estensione ".jar" è sufficiente inserirlo nella cartella plugins di Eclipse. È consigliato avviare Eclipse con il flag "-clean" in modo da partire da un base pulita.

8.8 Screenshot

Di seguito riporto uno screenshot dell'interfaccia del plugin Eclipse che mostra il nuovo tab dedicato alle eccezioni e l'esempio di un messaggio di errore.