Laurea Specialistica In Ingegneria Informatica

Università Di Roma “Sapienza”

Facoltà Di Ingegneria

Anno Accademico 2007/2008

Jamote

Java **Mo**del-Based Black-Box **Te**sting

Sistema per eseguire test a scatola nera basato
sulla costruzione di modelli astratti a partire da specifiche formali

Corso Di Metodi Formali Nell'Ingegneria Del Software

Docente: *Mancini Toni*

Studenti:

Casciaro Mario *Palleschi Andrea* *Profice Biagio*

Indice

1. Introduzione

[1.1] Introduzione al Model-based testing

Prima di entrare nel dettaglio del funzionamento del sistema viene di seguito illustrata l'idea alla base del test basato su modelli.

Un modello è una rappresentazione astratta del sistema sotto test. In particolare, è possibile rappresentare tramite modelli astratti sia il comportamento interno vero e proprio del sistema, sia i dati di input/output da utilizzare come casi di test. Il model-based black-box testing, ricade per forza di cose, in quest'ultimo caso. Per la generazione dei vari casi di test esistono varie tecniche, tutte basate sull'utilizzo di logica e/o linguaggi formali:

- casi di test generati attraverso **theorem proving**: theorem proving è utilizzato per la dimostrazione automatica di formule logiche. Con questo approccio il sistema è modellato come un insieme di formule logiche (predicati) che specificano il comportamento del sistema.
- test generati attraverso il **model checking**: al model checker viene passato il modello del sistema e una proprietà da testare. Durante la verifica della proprietà vengono individuati "testimoni" e "controesempi" che non sono altro che dei percorsi all'interno del grafo di esecuzione. Tali percorsi possono essere utilizzati come casi di test da eseguire direttamente sul sistema.
- casi di test generati attraverso il **model finding**: Viene definita una rappresentazione formale delle specifiche del sistema. Successivamente vengono generati i modelli di tali specifiche, che poi vengono utilizzati come casi di test.

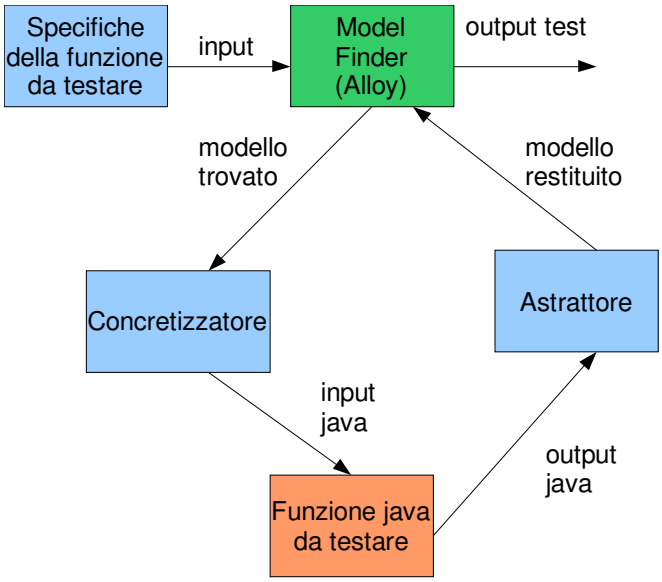
Come si è già detto prima nel model-based testing, gli input/output della funzione sono delle rappresentazioni astratte dei valori reali. Questo rende necessaria la definizione di un mapping tra il modello astratto e i dati reali, questi ultimi direttamente utilizzabili come casi di test.

Una volta generati i modelli vi sono vari modi per eseguire i casi di test:

- **test online**: il test viene eseguito dinamicamente sul sistema dal tool per la generazione dei modelli di test,il quale si connette direttamente ad esso.
- **generazione offline di casi di test eseguibili automaticamente**: il tool per la generazione dei modelli di test genera dei casi di test in una forma leggibile dall'applicazione da testare, questi possono essere in seguito eseguiti automaticamente (es. per un'applicazione Java vengono create delle classi Java che rappresentano i casi di test).
- **generazione offline di casi di test eseguibili manualmente**: il tool per la generazione dei modelli genera dei casi di test in una forma leggibile dall'uomo, ad esempio un file di testo in cui vengono descritti i casi di test, il test in questo modo deve essere eseguito manualmente sull'applicazione da testare.

[1.2] Breve descrizione dell'applicazione

Il sistema ha lo scopo di generare automaticamente casi di test a scatola nera per funzioni Java al fine di verificarne la correttezza. L'esecuzione dell'applicazione è divisa in vari passi. Una volta ricevuti in input, la funzione Java da testare e le espressioni in sintassi Alloy che descrivono le specifiche del test da eseguire, l'applicazione attraverso Alloy genera dei modelli in accordo a tali specifiche che, successivamente attraverso un processo di "concretizzazione" vengono convertiti in input per la funzione da testare; viene eseguita la funzione con tale input e una volta ottenuto l'output, attraverso un processo di "astrazione" viene ricreato un modello per Alloy il quale verifica se esso rispetta la specifiche di output. La tipologia di esecuzione del test è dunque quella del test online e i casi di test vengono generati automaticamente da Alloy che attraverso un processo di model finding restituisce modelli coerenti con le specifiche di test.



[1.3] Descrizione dettagliata degli input all'applicazione

- Funzione Java da testare, già compilata.
- Precondizioni del test: tutte le possibili tipologie di input che una funzione non è in grado di gestire, ovvero per i quali, si sa che l'output risulterebbe non corretto. Le precondizioni vengono distinte dalle specifiche di input, poiché le prime cercano di evitare delle condizioni in cui la funzione darebbe un output imprevisto e che quindi non è possibile testare, mentre le seconde definiscono un insieme di valori di input di una classe di equivalenza all'interno dell'insieme definito dalle precondizioni.
- Classi di equivalenza del test: si suddividono in specifiche di input e specifiche di output. Le prime modellano tutti valori che la funzione da testare deve ricevere mentre le seconde modellano i valori che, per essere corretto, il sistema deve restituire in corrispondenza di tali input.
- Dimensione massima del modello generato: dimensione massima del modello generato da Alloy per testare una determinata partizione di equivalenza.

- Dimensione massima degli interi: espressa in numero di bit, limita il valore degli interi rappresentabili nel modello astratto. Ad esempio se la dimensione è n i possibili valori vanno da – ([2^(n-1)]-1) a [2^(n-1)]-1.
- Numero di test: numero massimo di casi di test (ovvero modelli) generati per la partizione di equivalenza data.
- Funzioni e predicati Alloy: viene data l'opportunità all'utente di specificare costrutti personalizzati che possono poi essere utilizzati per definire le precondizioni e le specifiche di input/output.

2. Progettazione e implementazione

[2.1] Funzionalità implementate

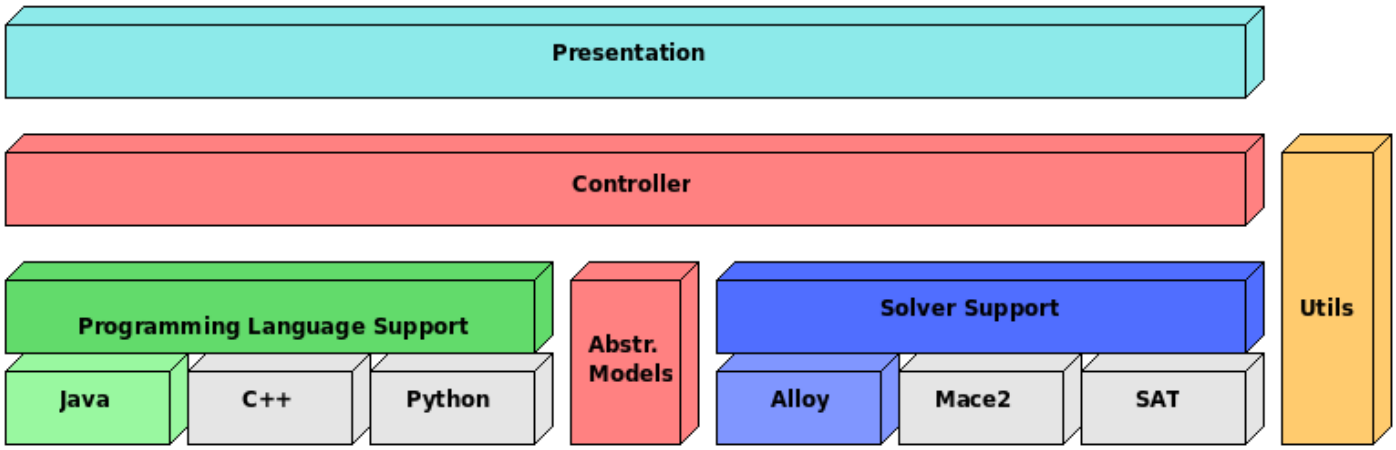
- **Generazione automatica dei casi di test a partire da una rappresentazione formale dell'input.** Jamote prende in input una formula espressa in un linguaggio formale e genera per mezzo di un “model finder” tutti i modelli di tale formula.
- **Esecuzione automatica dei test e verifica della correttezza confrontando l'output concreto con una sua specifica formale.** A partire da un input rappresentato sotto forma di modello astratto, Jamote è in grado di concretizzare tale input, e automaticamente fornirlo alla funzione da testare. L'output della funzione viene a sua volta trasformato automaticamente in un modello astratto e per mezzo del “model finder” viene verificato che tale astrazione sia un modello della formula che rappresenta la specifica di output della funzione.
- **Generazione automatica dell'output corretto a partire dalla specifica formale.** Nel caso l'output della funzione testata non sia un modello valido della specifica di output, Jamote è in grado di sfruttare tale specifica formale per generare un modello corrispondente all'output corretto atteso dalla funzione.
- **Supporto alle classi di equivalenza.** Jamote permette di definire diverse classi di equivalenza ognuna con la sua specifica di input e output. Si può così condurre una serie di test “mirati” sulle varie classi di input, permettendo di fare un'inferenza statistica sulla correttezza della funzione: se l'output è corretto per alcuni valori di input appartenenti ad una classe di equivalenza, allora è molto probabile che lo sia per tutti i valori della stessa classe. Ovviamente, la probabilità tenderà ad 1 all'aumentare degli input testati. A tal proposito Jamote permette di specificare, per ogni classe di equivalenza, il numero di modelli di input da generare.
- **Tipi di dato supportati per l'input/output:**
  - **Interi**
  - **Array di Interi**
  - **Eccezioni**

  Jamote allo stato di sviluppo attuale è in grado di supportare soltanto i suddetti di tipi di dato essendo questi i più significativi per dimostrare il funzionamento base del black-box model-based testing. Il supporto alle eccezioni è stato implementato sopratutto per dimostrare l'uso della suddivisione basilare dell'input in classi di equivalenza “valide” e “non valide”.
- **Rimozione automatica degli input duplicati.** A volte è possibile che il model finder fornisca modelli effettivamente differenti che tuttavia corrispondono ad input isomorfi. Questo è il caso, ad esempio, in cui il modello contenga istanze di oggetti “isolati” che poi non vengono considerati da Jamote parte del modello di input. Questo aspetto, verrà chiarito successivamente.
- **Riconoscimento automatico dell'integer overflow.** Essendo solitamente i “model

finder" basati su SAT, è plausibile che per motivi prestazionali, vengano imposti dei limiti sul dominio degli interi supportati. Questo aspetto è gestito in automatico nella fase di generazione dei modelli di input, lo stesso non si può dire dell'output restituito dalla funzione, che verosimilmente può essere un qualsiasi intero all'interno del dominio supportato dal linguaggio di programmazione. Consideriamo un esempio in cui andiamo ad imporre che il dominio di input sia tra 1 e 3, e imponiamo che l'output per essere corretto deve cadere tra -5 e 5. Il maggiore dei domini è quello (-5,5), quindi specifichiamo un integer bitwidth di 4 (questo significa che Alloy è in grado di gestire solo interi compresi tra -7 e 7). A questo punto, noi sappiamo che valori di ritorno fuori dall'intervallo (-5,5) non sono corretti, però per dimostrarlo formalmente con Alloy dobbiamo avere un valore di output che sia compreso nel dominio supportato da Alloy (-7,7). Risulta quindi evidente, che se l'output fosse pari a 6 Alloy riuscirebbe a dimostrare la non correttezza, tuttavia se la funzione ritornasse 8, Alloy non sarebbe in grado di gestire tale valore e quindi non sarebbe in grado di dimostrare la non correttezza dell'output. In queste situazioni interviene Jamote, per segnalare all'utente l'impossibilità da parte di Alloy di dimostrare le proprietà dell'output. In realtà l'unico modo per evitare a priori situazioni di questo tipo sarebbe quello di specificare il dominio di Alloy uguale al dominio degli interi supportati dal linguaggio di programmazione. **Jamote, quindi, è in grado di riconoscere automaticamente e scartare (lanciando un messaggio di warning) se un intero va oltre il dominio supportato dal model finder.**

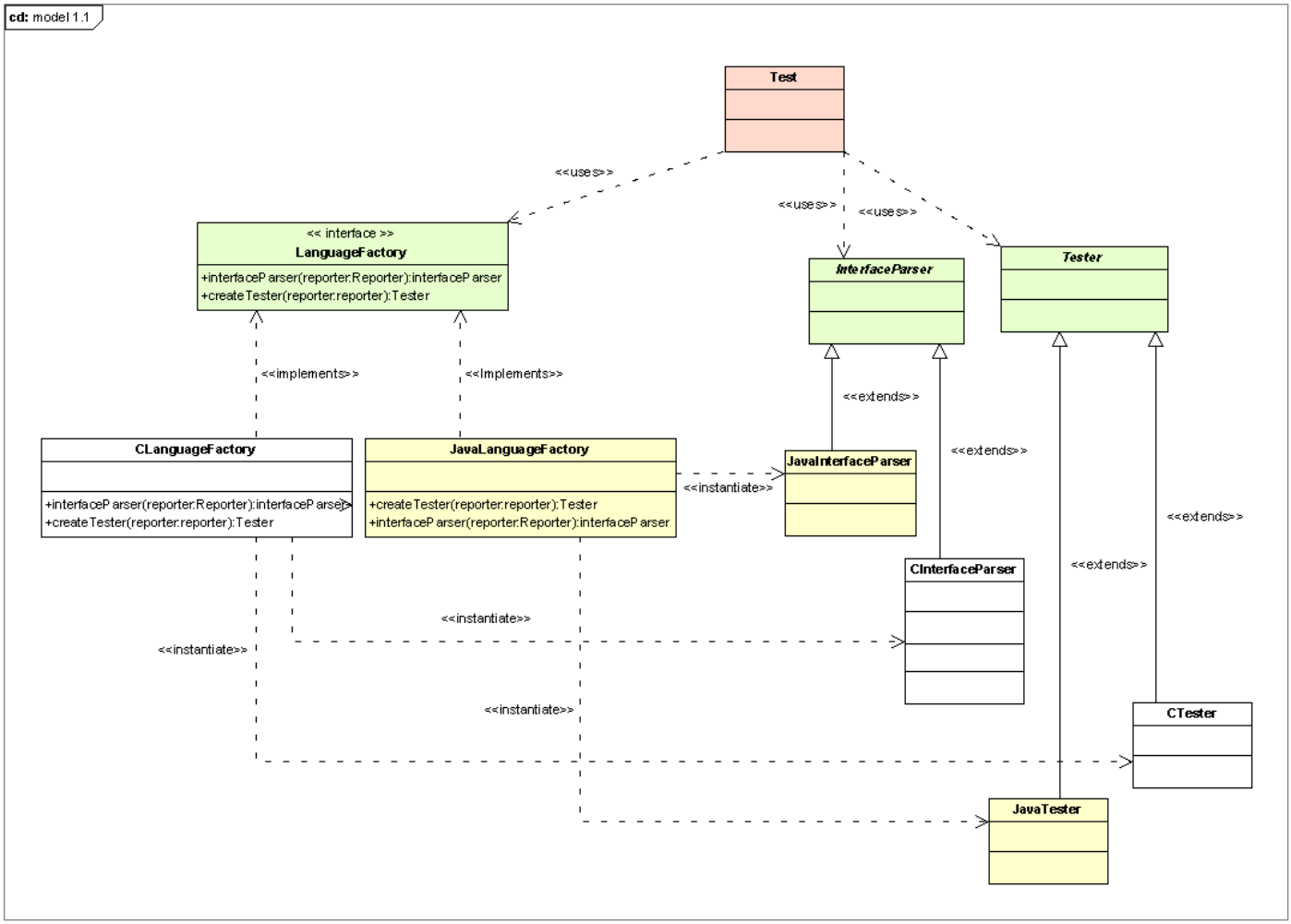
- **Possibilità di aggiungere il supporto ad altri linguaggi di programmazione e altri model finder.** La struttura estremamente modulare di Jamote permette di aggiungere con semplicità il supporto a diversi linguaggi di programmazione e/o tool per la generazione di modelli astratti. Semplicemente sostituendo gli opportuni moduli sarebbe possibile infatti testare funzioni Java, C++, Python o un altro linguaggio di programmazione, oppure utilizzare Alloy, Mace2, o direttamente SAT per generare i modelli, tutto questo senza modificare il core di Jamote.

[2.2] Struttura del sistema



Come si è già accennato, durante la progettazione di Jamote sono stati tenuti in forte considerazione i fattori modularità e riuso. Come si nota dallo schema qui sopra, ogni modulo all'interno del sistema è completamente disaccopiato dagli altri moduli. In particolare è presente un disaccoppiamento sia verticale sia orizzontale all'interno dello stack del sistema. Questo vuol dire ad esempio che il modulo “Solver support” è disaccoppiato sia dal “Controller” sia dal “Programming language Support”.

Il **disaccoppiamento verticale** verso l'alto è una caratteristica strutturale di base in un sistema ben progettato (ovvero è ovvio che il modulo “Solver support” sia completamente disaccoppiato da chi lo utilizza, in questo caso il “Controller”) e quindi non ci soffermeremo su tale aspetto. Un discorso a parte merita il disaccoppiamento verticale verso il basso. Questa è una caratteristica molto utile per aumentare la riusabilità generale del sistema e solitamente è ottenibile tramite l'utilizzo di classi astratte e/o interfacce. In particolare in Jamote i moduli “Programming Language Support” e “Solver Support” sono moduli astratti, questo permette di “sostituire”, anche a runtime, le implementazioni di tali moduli senza che il resto del sistema si accorga della differenza. Tale operazione è resa possibile grazie all'utilizzo del pattern “Abstract Factory”.

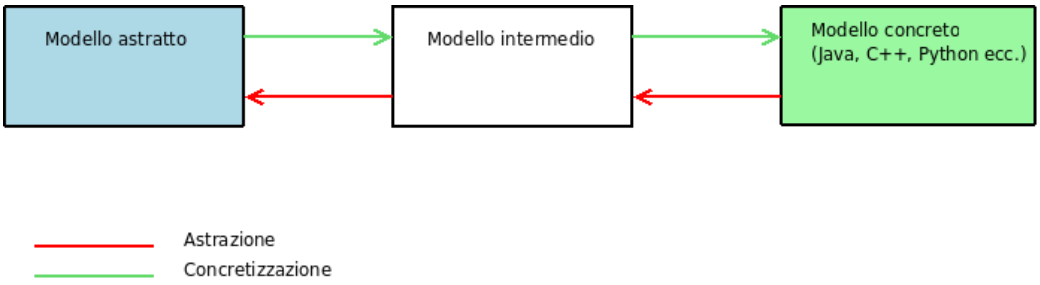


L’abstract Factory fornisce un’ interfaccia che consente di creare famiglie di oggetti connessi tra loro in modo che il client dell’applicazione non abbia necessità di riferirsi direttamente alla loro implementazione all’interno del proprio codice. In questo modo si permette che un

sistema sia indipendente dall'implementazione degli oggetti concreti e che il client, attraverso l'interfaccia, possa utilizzare diverse tipologie di oggetti. Nella figura qui sopra viene riportata la struttura del pattern così come è stato applicato al modulo “Programming Language Support”. Si noti, come la classe “Test” che è il client del pattern deve solo scegliere l'implementazione del “Factory” dopodiché ogni istanza restituita dal tale factory sarà coerente con l'implementazione scelta. Il client, dunque, può riferirsi sempre e solo, sia al factory sia agli oggetti da esso creati, tramite le loro interfacce, e può quindi non conoscere affatto la loro implementazione, anche se la sta effettivamente utilizzando. Di seguito si riporta uno spezzato del codice della classe Test:

```
[...]
languageFactory = new JavaLanguageFactory();
[...]
modelingFactory = new AlloyModelingFactory();
[...]
Solver solver = modelingFactory.createSolver(this.reporter);
Tester tester = languageFactory.createTester(this.reporter);
[...]

//Per ogni classe di equivalenza
for(EquivalencePartition partition :partitions){
	[...]
	List<InputModel> models = solver.generateModels(function,defines,precoditions,
partition);
	for(InputModel inputModel: models){
		[...]
		OutputModel outputModel = tester.executeTest(function, inputModel);
		int successful = solver.checkOutput(function, defines,
					partition, inputModel, outputModel);
		[...]
	}
}
```

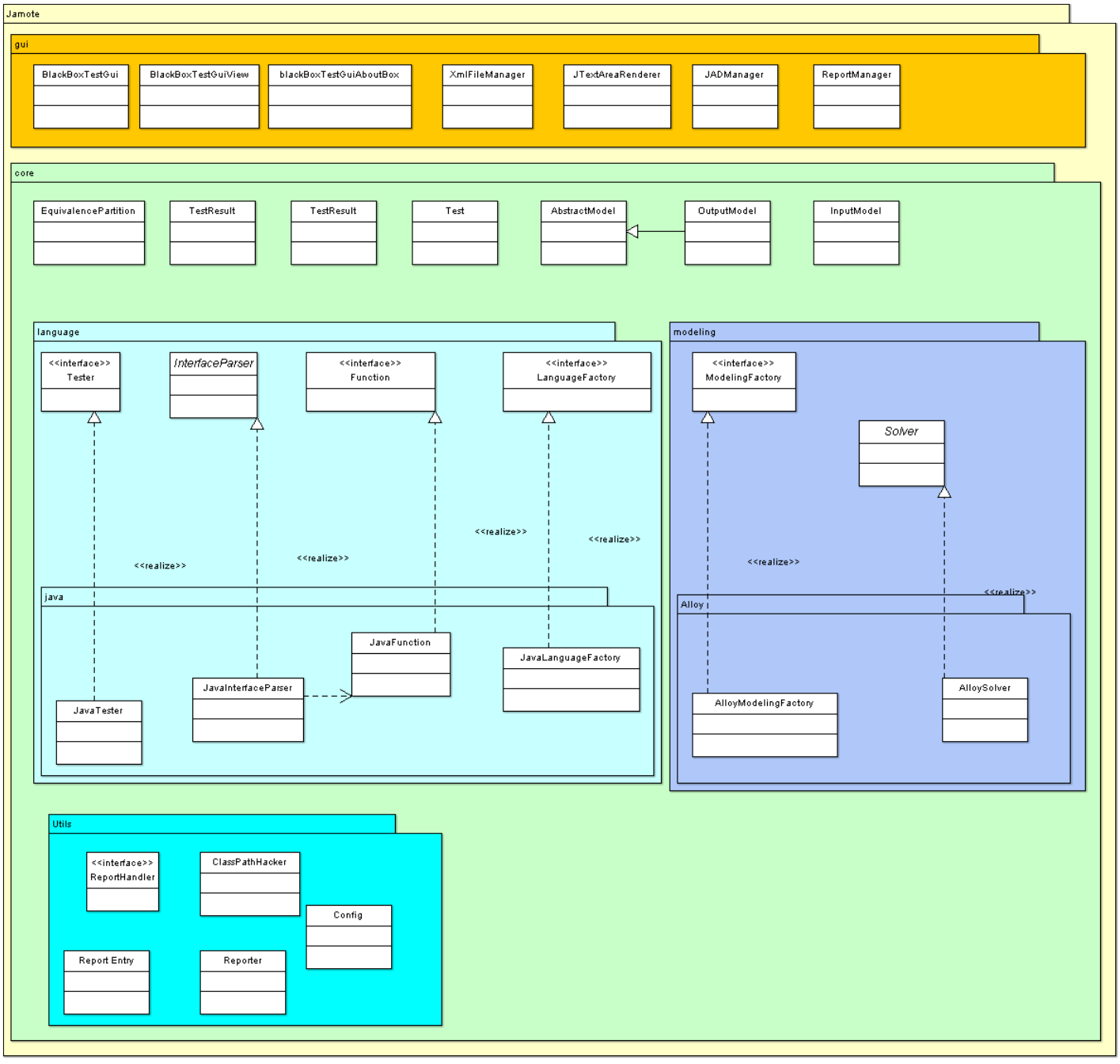
Il **disaccoppiamento orizzontale** in Jamote è stato introdotto per separare tra loro i moduli “Programming Language Support” e “Solver Support”. Questo risultato è stato ottenuto inserendo un modulo intermedio, contenente in particolare tutte le classi utilizzate dai due moduli per scambiarsi tra loro i dati. Queste classi non sono niente altro che una rappresentazione semi-concretizzata dei modelli di input/output. Questo significa che sia il processo di concretizzazione sia quello di astrazione avverranno in due fasi differenti, passando sempre per il modello intermedio.



In particolare, la prima fase del processo di astrazione è a carico del “Programming language support” mentre la seconda fase del modulo “Solver Support”. La concretizzazione, invece, segue il percorso opposto. Il beneficio di questa tecnica è

evidente, infatti, nel processo di concretizzazione ad esempio, non è necessario sapere a priori il tipo di modello concretizzato da ottenere, sia esso un insieme di dati Java, o C++, tale passo sarà a carico del modulo specifico del linguaggio e quindi totalmente disaccoppiato dal resto del sistema.

[2.3] Class Diagram



E' riportato qui sopra il diagramma delle classi semplificato (senza associazioni e dipendenze) dell'applicazione.

Descrizione dei vari package dall'applicazione

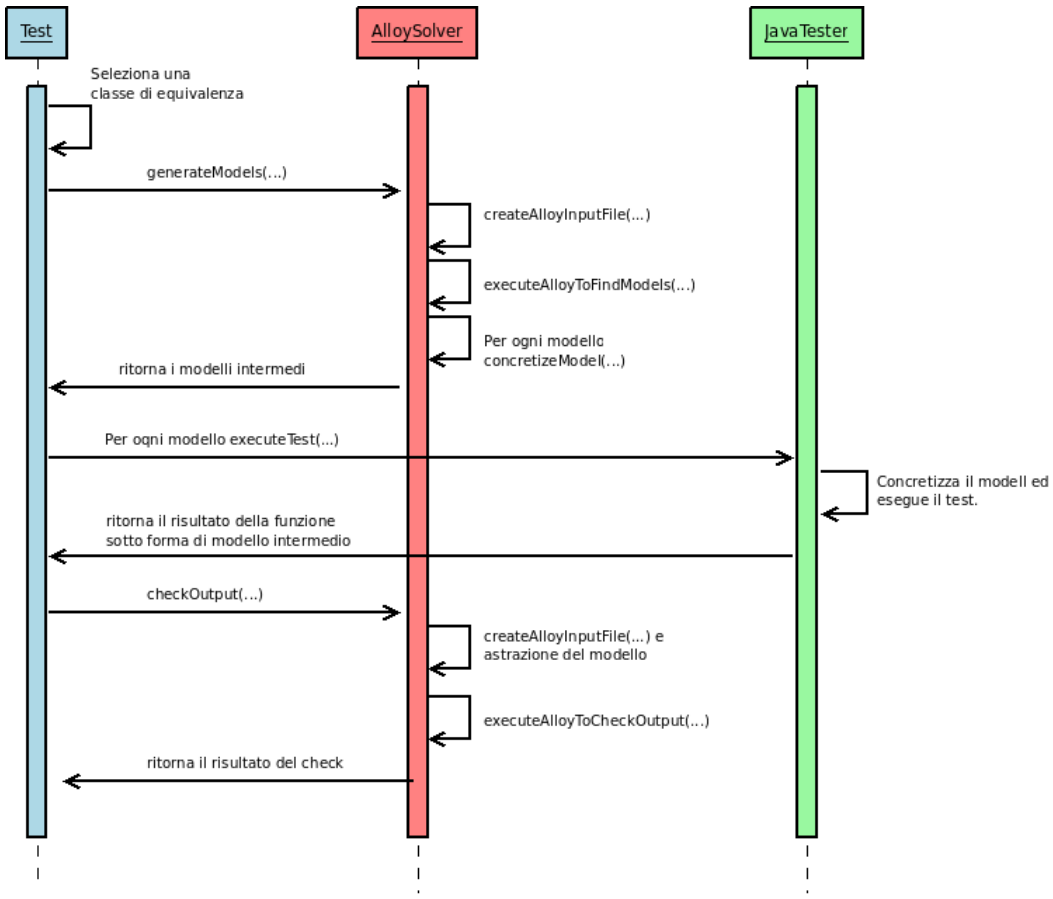
- “**gui**” package: contiene le classi java che si occupano della gestione dell’interfaccia grafica e della relativa logica di presentazione.

- “**core**” package: racchiude tutte le classi e i package che realizzano le funzionalità “chiave” dell'applicazione. In questo livello sono anche definite le classi che rappresentano il “modello intermedio” già descritto in precedenza, nonché il controller dell'intera applicazione (implementa il pattern “Facade”) ovvero la **classe “Test”**. Tale classe rappresenta l'unico punto di accesso all'intero core dell'applicazione, nascondendo la complessità interna del sistema. In pratica tale classe fornisce il metodo:

```
public Map<String, List<TestResult> > executeTest(
     String precoditions,
     List<EquivalencePartition> partitions,
     String defines, Function function)
    throws JamoteException
```

  grazie al quale è possibile invocare in modo immediato l'esecuzione di un test, semplicemente passando a tale metodo tutti gli input necessari all'esecuzione del test.
- “**core.language**” package: contiene le interfacce e le classi astratte che modellano funzionalità dell'applicazione che dipendono strettamente dal linguaggio di programmazinoe della funzione da testare.
- “**core.language.java**” package: contiene le classi che implementano le interfacce e classi astratte del package “language”, tale implementazione è in grado di gestire funzioni da testare scritte in linguaggio Java.
- “**core.modeling**” package: contiene le interfacce e classi astratte che si occupano dell'interazione con il model finder e che quindi dipendono strettamente da esso.
- “**core.modeling.alloy**” package: fornisce un'implementazione delle classi e delle interfacce in “core.modeling”. Tale implementazione permette l'interfacciamento con il model finder Alloy Analyzer.
- “**core.utils**” package: contiene classi d’utilità generale per l’applicazione.

[2.4] L'esecuzione del test



In questo diagramma viene riportato lo schema semplificato dell'esecuzione di un test.

1. Come prima cosa viene selezionata una classe di equivalenza da testare tra quelle definite dall'utente.
2. Tale classe di equivalenza viene passata all'oggetto AlloySolver per generare i modelli di input.
3. L'oggetto AlloySolver crea il file di input da passare ad Alloy, esegue l'AlloyAnalizer, prende il risultato e lo concretizza nel modello intermedio.
4. L'oggetto Test ora, passa tali modelli intermedi all'oggetto JavaTester che esegue l'ultima fase di concretizzazione del modello ed invoca la funzione da testare, passando come input tale modello.
5. Il risultato dell'esecuzione viene astratto in un modello intermedio dall'oggetto JavaTester e ritornato all'oggetto Test.
6. Tale modello viene dunque passato nuovamente ad AlloySolver che provvederà ad astrarlo in un modello compatibile con Alloy e verificherà la correttezza del risultato a fronte delle specifiche di output fornite dall'utente.
7. Infine il risultato di tale verifica viene ritornato all'oggetto Test.

Nel caso la verifica del risultato avesse esito negativo, l'esecuzione continuerebbe, in particolare l'oggetto Test chiederebbe ad AlloySolver di generare il risultato atteso per la funzione.

[2.5] Dettagli implementativi

[2.5.1] Interfacciamento con Alloy, astrazione, concretizzazione

Jamote utilizza Alloy **integrandosi direttamente con le sue API**. In particolare, per facilitare il debug, viene fornito l'input ad Alloy sotto forma di un normale file di testo. Il processo di **astrazione** dell'output della funzione avviene proprio in questa fase, in cui il modello dell'output viene convertito in un formato compatibile con la sintassi Alloy e inserito nel file di input passato all'Alloy Analizer. La struttura di questi file verrà analizzata in dettaglio successivamente.

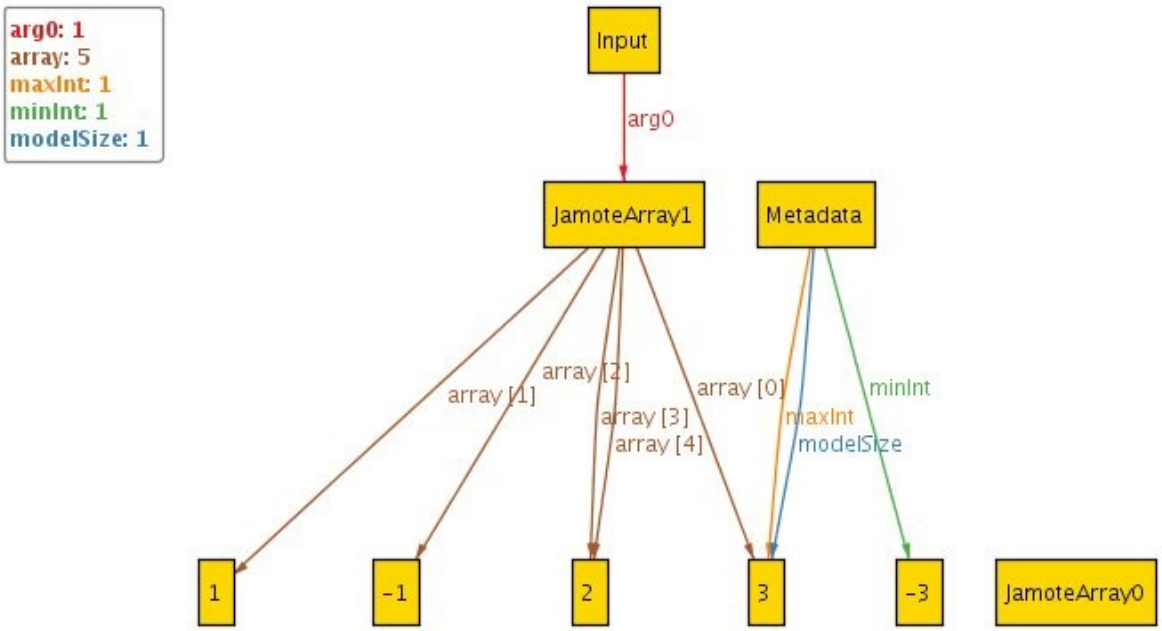
Per quanto riguarda i modelli generati da Alloy, questi vengono estratti direttamente dagli oggetti A4Solution che vengono restituiti come risultato della computazione. Di seguito viene riportato uno spezzato del codice che esegue il parsing del file Alloy, esegue i comandi, estrae il risultato e lo passa ad un'altra funzione di Jamote per processarlo.

```
Module world = CompUtil.parseEverything_fromFile([...], file.getPath());

//Per ogni comando
for (Command command: world.getAllCommands()) {
       //Eseguo il comando
       A4Solution ans = TranslateAlloyToKodkod.execute_command([...], command, [...]);

      //Processo le soluzioni se soddisfacibili
       for(int i = 0; i < partition.getMaxNumberOfTests();){
                     if(! ans.satisfiable())
                            break;

                     InputModel m = processModel(ans, [...]);
                     [...]
       }
       [...]
}
```

E' utile far notare che Alloy genera ogni **modello sotto forma di grafo** in cui ogni nodo rappresenta l'instanza di una sig e gli archi rappresentano le associazioni tra tali istanze. Jamote andrà quindi ad eseguire il processo di concretizzazione eseguendo il parsing di tale grafo estraendo tutte e sole le informazioni del modello di cui ha bisogno. Facciamo un esempio per chiarire questo aspetto. Supponiamo di avere una funzione che prende in input un array di interi, e consideriamo una specifica di input di questo tipo, in cui imponiamo che la dimensione dell'array sia 5 che ogni intero al suo interno sia nell'intervallo (minInt,MaxInt):

```
all i:Int | input.arg0.at[i] <= metadata.maxInt && input.arg0.at[i] >= metadata.minInt) &&
input.arg0.length = 5
```

Un possibile modello generato da tale formula avrà tale struttura:



Notiamo come anche gli interi siano considerati dei nodi. Jamote a questo punto visiterà il grafo a partire dal nodo “Input” per poi visitare, percorrendo gli archi arg0...argN tutti gli argomenti del modello di input. Nel caso un argomento fosse un array Jamote visiterà ogni arco che modella l'associazione Array->indice->valore. Inoltre, per ogni argomento Jamote effettuerà il processo di **concretizzazione** ovvero trasformerà i dati rappresentati da Alloy in dati Java concreti utilizzabili per testare la funzione. Questa ad esempio è la funzione che concretizza un array di interi nella struttura concretizzata intermedia:

```
	private AbstractObject concretizeIntArrayTuple(A4Solution ans, Field f)
			throws JamoteModelingException,Err
	{
		//Facciamo un check sul tipo
		Type t = f.type;
		List<List<PrimSig>> psigs = t.fold();
		if(t.arity() != 2)
			throw new JamoteModelingException("Unexpected argument type in abstract
model," +
					"arity is != 2, JamoteArray expected");

		PrimSig ps1 = psigs.get(0).get(1);
		if(!ps1.label.equals("this/JamoteArray"))
			throw new JamoteModelingException("Unexpected argument type" +
					"["+ps1.label+ "]" +
					"in abstract model, JamoteArray expected");

		//Ora che il tipo è corretto estraiamo il valore dell'argomento
		A4TupleSet tupleSet = (A4TupleSet) ans.eval(f);

		//l'argomento è null
		if(!tupleSet.iterator().hasNext())
			return new AbstractObject("int[]","int[]",null);
		//la prima sig è quella che rappresenta la Sig padre es. Output$0
		//la seconda deve essere un Array
		PrimSig arraySig = tupleSet.iterator().next().sig(1);
		String sigLabel = tupleSet.iterator().next().atom(1);

		Map<Integer, Integer> arrayMap = new HashMap<Integer, Integer>();
		Field arrayField = arraySig.getFields().iterator().next();
		tupleSet = (A4TupleSet) ans.eval(arrayField);
```

```
            for(A4Tuple valueTuple: tupleSet){
                  if(valueTuple.atom(0).equals(sigLabel))
                        arrayMap.put(new Integer(valueTuple.atom(1)), new
Integer(valueTuple.atom(2)));
            }

            //Ora trasformiamo la Map in un array
            int ret[]  = new int[arrayMap.size()];
            for(int i = 0; i < arrayMap.size(); i++)
                  ret[i] = arrayMap.get(new Integer(i)).intValue();

            return new AbstractObject("int[]","int[]",ret);
      }
```

Ritornando all'immagine del modello di esempio, mostrata sopra, possiamo notare il **problema delle "istanze isolate"** accennato precedentemente. In questo caso l'istanza "JamoteArray0" è isolata, e quindi non verrà inclusa nel modello di input utilizzato per testare la funzione. A questo punto potremmo avere due modelli identici tranne che per la presenza o meno dell'istanza isolata, tali modelli sono giustamente considerati differenti da Alloy ma ai fini della nostra applicazione sono del tutto indistinguibili. Questa situazione può essere evitata con qualche accortezza da parte di chi scrive le specifiche, tuttavia non sempre il problema viene identificato dall'utente, soprattutto per il fatto che Jamote non da l'output del modello completo, né tanto meno ne fornisce una rappresentazione visuale. Per rendere più semplice e immediato l'utilizzo del tool, nonché per evitare a priori la presenza di modelli duplicati, Jamote implementa un meccanismo per scartare tali modelli e non includerli affatto tra quelli utilizzati per testare la funzione.

[2.5.2] Struttura dei file di input

Jamote utilizza 3 strutture diverse per i file di input da passare ad Alloy, una per ogni diversa operazione, ovvero "generazione modelli", "verifica dell'output", "generazione dell'output atteso".

Di seguito descriviamo la struttura generale di tali file. In questa sezione, oltre ad utilizzare la sintassi Alloy, verrà utilizzata una notazione particolare per esprimere alcuni costrutti inseriti a runtime da Jamote nel codice.

- *$[Variabile]* : con questa notazione verranno indicate le varibili che Jamote definisce al suo interno e che inserisce negli script Alloy generati.
- *?[String]* : indica una stringa che Jamote inserisce all'interno del codice Alloy solo sotto certe condizioni.

[2.5.3] File per la "generazione dei modelli"

Questo è il file grazie al quale Jamote utilizza Alloy per generare i modelli di input. Di seguito

si riporta la sua struttura generale.

1. *Strutture Jamote di supporto*

In questa sezione viene dichiarata una serie di sig e funzioni/predicati che mettono a disposizione gli strumenti necessari a Jamote per funzionare correttamente e utili all'utente durante la specifica delle formule Alloy. In particolare vengono definiti:

- **Un header**, in cui vengono importati vari file esterni di utilità, in particolare se nell'interfaccia della funzione viene specificato il tipo “int”, automaticamente viene importato il file utils/integer:

```
open util/integer as mod_int
```

Se viene specificato il tipo “int[]” viene importato il file utils/sequniv:

```
open util/sequniv as mod_sequniv
```

- **Una sig Metadata**, utilizzata per rendere disponibile all'interno delle formule Alloy utilizzate per le specifiche, i dati relativi all'esecuzione corrente del solver, ovvero il massimo e il minimo intero supportato e la dimensione massima del modello:

```
one sig Metadata
{
   maxInt: one Int,
   minInt: one Int,
   modelSize: one Int
}{
   maxInt= $[MaxInt] &&
   minInt= $[MinInt] &&
   modelSize= $[ModelSize]
}
```

- **Una sig JamoteArray**, definita nel caso si richieda il supporto al tipo “int[]” e che rappresenta la sig utilizzata per il “wrapping” di tale tipo di dato. Ogni array di interi, definito all'interno di tutto il file Alloy di input così come le specifiche formulate dall'utente, andranno ad utilizzare tale sig ogni volta che ci si riferirà ad un tipo “int[]”. Vengono inoltre definite una serie di funzioni e predicati utili per manipolare tale sig.

```
sig JamoteArray{
   array: seq Int
}

fun at[a: JamoteArray, i:seq/Int]: Int{
   a.array[i]
}

fun length[a: JamoteArray]: Int{
```

```
      #a.array
   }

   pred hasIndex[a: JamoteArray, idx: seq/Int]{
      idx < #a.array && idx >= 0
   }
```

2. *Strutture “core” di Jamote*

In questa sezione sono definite le strutture “chiave” utilizzate da Jamote per rappresentare il modello da generare. In particolare per permettere di generare i modelli di input viene definita una **sig Input** che rappresenta appunto gli argomenti da passare in input alla funzione. Tale sig andrà quindi a rispecchiare i tipi di dato accettati in input dalla funzione da testare, nell'ordine in cui sono stati dichiarati.

```
sig Input
{
   arg0: $[multiplicity] $[Type0]
   ...
   argN: $[multiplicity] $[TypeN]
}
```

3. *Strutture di supporto definite dall'utente*

A questo punto del file di input vengono inserite tutte le funzioni, predicati, sig definite dall'utente, in modo da renderle disponibili nel resto del file.

4. *Fact per le precondizioni*

Le precondizioni specificate dall'utente vengono scritte in un apposito fact e andranno quindi a rappresentare un vincolo sempre valido durante la generazione dei modelli.

```
fact preconditions
{
    ?[all input:Input|] ?[all metadata:Metadata|] $[preconditions]
}
```

5. *Pred per le specifiche di input*

Le specifiche di input della classe di equivalenza da testare vengono scritte in un apposito predicato, successivamente per permettere la generazione dei modelli si richiederà ad Alloy di cercare tutti i modelli che rendono vero tale predicato.

```
pred inputSpecs
{
    ?[all input:Input|] ?[all metadata:Metadata|] $[input specification]
}
```

6. *Comando*

Per finire, viene specificato il comando grazie al quale si indica il predicato da

verificare nella ricerca dei modelli e viene imposta ad Alloy la dimensione di tali modelli.

```
run inputSpecs for $[modelDimension] but $[integerBitwitdh] int, exactly 1 Input,
exactly 1 Metadata
```

[2.5.4] File per la “verifica dell'output”

Questo è il file di input che Jamote genera per consentire ad Alloy la verifica dell'output di una funzione. Tale file è così strutturato:

1. ***Strutture Jamote di supporto***

   Oltre alle strutture **header, sig Metadata, sig JamoteArray** già viste in precedenza, viene definita un'ulteriore sig chiamata **Exception** che rappresenta appunto un'eccezione lanciata dalla funzione da testare.

```
sig Exception  {}
```

   Inoltre viene specificata una sig per ogni tipo di eccezione lanciata dalla funzione da testare, avente come nome il nome di tale eccezione. Se ad esempio la funzione da testare dichiara nel campo “throws” l'eccezione “IOException” avremo la generazione di una sig IOException.

```
sig IOException extends Exception {}
```

   Inoltre se la funzione lancia a runtime un'eccezione non specificata nel campo “throws” ad esempio “NullPointerException”, Jamote genererà automaticamente una sig anche per questo tipo di eccezione (anche se tale sig non potrà di fatto essere utilizzata nelle specifiche di output poiché viene generata solo nel caso essa vanga effettivamente lanciata).

2. ***Strutture “core” di Jamote***

   Oltre alla **sig Input** già analizzata precedentemente, avremo in questa sezione la definizione della **sig Output** che rappresenta appunto l'output della funzione da testare, tale sig conterrà un campo “return” che rappresenta il valore di ritorno della funzione e un campo “exception” che conterrà l'eventuale eccezione lanciata.

```
sig Output
{
   return: lone $[returnType],
   exception: lone Exception
}
```

3. ***Strutture di supporto definite dall'utente***
4. ***Fact per la specifica dell'input astratto***

   In questa sezione viene inserito un fact che ponendo dei vincoli sulla sig Input, andrà a specificare i valori con i quali la funzione è stata testata. Tale fact definisce a tutti gli effetti una **rappresentazione astratta dell'input** della funzione. Consideriamo il semplice caso in cui la funzione da testare abbia un solo argomento di tipo intero e che il valore assegnato a tale argomento sia stato 3, Jamote genererà un fact di questo tipo:

```
fact inputValues
{
    all i:Input | i.arg0 = 3
}
```

   Di seguito si riporta come esempio il codice Java utilizzato per astrarre un array di interi:

```
String ret += "(";
if(o.getValue() == null)
{
      ret += "(all r:" + OutputSig + " | no u:univ | r." + RetField + "=u)\n";
}
else
{
      int[] arr = (int[]) o.getValue();
      ret += "all r:" + OutputSig + " | r."+ RetField +".length = " + arr.length;
      if(arr.length > 0)
            ret += " && ";
      for(int j = 0; j < arr.length; j++){
            ret += "r."+ RetField +".at["+j+"]="+arr[j] + " ";
            ret += " && ";
      }
      ret += "(all r:Output | one u:univ |r.return = u)";
}

ret += ")\n";
```

5. ***Fact per le specifiche dell'output astratto***

   Analogamente alla sezione precedente, verrà creato un fact per specificare il valore di ritorno della funzione testata. In particolare avremo due casi, uno in cui la funzione ritorni effettivamente un valore e uno invece in cui la funzione lanci un'eccezione. Analizziamo con un esempio, le strutture generate in entrambi i casi. Consideriamo che la funzione ritorni con successo un intero pari a 3, avremo che verrà generato un fact del tipo:

```
fact outputValue
{
    all o:Output | o.return = 3 && (no u:univ | o.exception = u)
}
```

Dove si impone che il valore di ritorno sia pari a 3 e che non ci siano eccezioni nel campo Output.exception.

Supponiamo ora che la funzione lanci un'eccezione del tipo IOException, avremo la generazione di un fact di questa forma:

```
fact exceptionValue
{
   (all o:Output | one e:IOException| e=o.exception) &&
   (all o:Output | no u:univ | o.return = u)
}
```

In questo caso viene imposto che nel campo “return” non vi sia alcun valore, e che nel campo “exception” vi sia un oggetto di tipo IOException.

6. ***Assert per le specifiche di output***

   A questo punto, viene creata un'asserzione per inserire le specifiche di output della funzione così come sono state formulate dall'utente. Tale asserzione verrà utilizzata da Alloy per verificare che l'input e l'output astratti precedentemente specificati siano effettivamente un modello della specifica di output della funzione.

```
assert outputSpecs
{
   ?[all output:Output|] ?[all metadata:Metadata|] ?[all input:Input|]
   $[outputSpecification]
}
```

7. ***Comando***

   Per completare il file abbiamo bisogno del comando per istruire Alloy sulla asserzione da verificare e la dimensione del modello.

```
check outputSpecs for $[modelSize] but $[integerBitwidth] int, exactly 1 Input,exactly
1 Output ,1 Exception, exactly 1 Metadata
```

[2.5.5] File per la “generazione dell'output atteso”

Con questo file Jamote utilizza Alloy per ricavare un modello delle specifiche di output. Questo permette di ottenere l'output della funzione, atteso dall'utente. Ricordiamo che questa fase ha luogo solo se la precedente fase di verifica dell'output è terminata con esito negativo.

1. ***Strutture Jamote di supporto***
2. ***Strutture “core” di Jamote***
3. ***Strutture di supporto definite dall'utente***
4. ***Fact per la specifica dell'input astratto***

Tutte le sezioni qui sopra specificate sono del tutto uguali a quelle del file per la “verifica dell'output”

5. ***Pred per le specifiche di output***

   In modo analogo al file per la “verifica dell'output” vengono incluse le specifiche dell'output formulate dall'utente, questa volta però non vengono incluse in una asserzione ma in un predicato (pred expectedOutput) con cui andremo ad invocare il comando “run” di Alloy che genererà un modello corretto di tali specifiche.

6. ***Comando***

   Alla fine del file viene inserito il comando per invocare la computazione Alloy mirata alla ricerca dei modelli che rappresentano l'output corretto della funzione.

```
run expectedOutput for $[modelSize] but $[integerBitwidth] int, exactly 1
Input,exactly 1 Output ,1 Exception, exactly 1 Metadata
```

3. Manuale d'uso

[3.1] Requisiti

I requisiti per l'esecuzione dell'applicazione sono:

Java Runtime Environment Versione 1.6

Il programma è stato testato sulle seguenti piattaforme:

- Windows XP, Windows Vista
- Linux Ubuntu 8.10
- Mac Os X Leopard

Essendo scritto in Java, non si esclude il funzionamento su altri sistemi operativi, purché si utilizzi la versione 1.6 del JRE. E' stata anche sviluppata una versione per JRE 1.5, per soli fini di compatibilità, tuttavia non è presente la funzione di decompilazione del codice.

La funzionalità di decompilazione del codice è invece supportata solo nei sistemi operativi sopraelencati anche se non si esclude il funzionamento anche su altre distribuzioni Linux e versioni di Mac OS.

[3.2] Sintassi per la specifica dell'input e dell'output

Per la specifica dell'input e dell'output sono ammessi tutti i costrutti presenti nella sintassi Alloy, con l'aggiunta della definizione di alcuni costrutti, utilizzabili durante la scrittura delle specifiche di input e di output.

- Per riferirsi all'input e ai suoi argomenti Jamote mette a disposizione la **variabile "input"** che conterrà un'istanza della sig Input precedentemente definita. Si può dunque accedere agli argomenti della funzione tramite i campi arg0...argN. Ad esempio se si volesse limitare tra -2 e 2 la dimensione del primo argomento, si potrebbe scrivere nelle specifiche di input:

```
input.arg0 >= -2 && input.arg0 <= 2
```

- Analogamente, per riferirsi al valore di ritorno della funzione, viene ,messa a disposizione la **variabile "output"** che conterrà un'istanza della sig Output. Quindi, il valore vero e proprio della funzione (se presente) si troverà nel campo "output.return" mentre se la funzione sarà terminata con un'eccezione ci si può riferire ad essa con "output.exception". Ovviamente tali variabili sono disponibili soltanto nelle specifiche di output.
- Inoltre è possibile accedere ad una serie di informazioni sull'esecuzione del programma grazie alla **variabile "metadata"** che conterrà un'istanza della sig

Metadata. Tale sig (come già descritto in precedenza) conterrà i campi:

- metadata.maxInt: conterrà il valore del massimo intero gestibile da Alloy.
- metadata.minInt: conterrà il valore del minimo intero gestibile da Alloy.
- metadata.modelSize: conterrà la dimensione massima del modello che Alloy dovrà cercare.

[3.3] Descrizione GUI e funzionamento

Di seguito verrà descritta dettagliatamente la GUI e il suo layout.

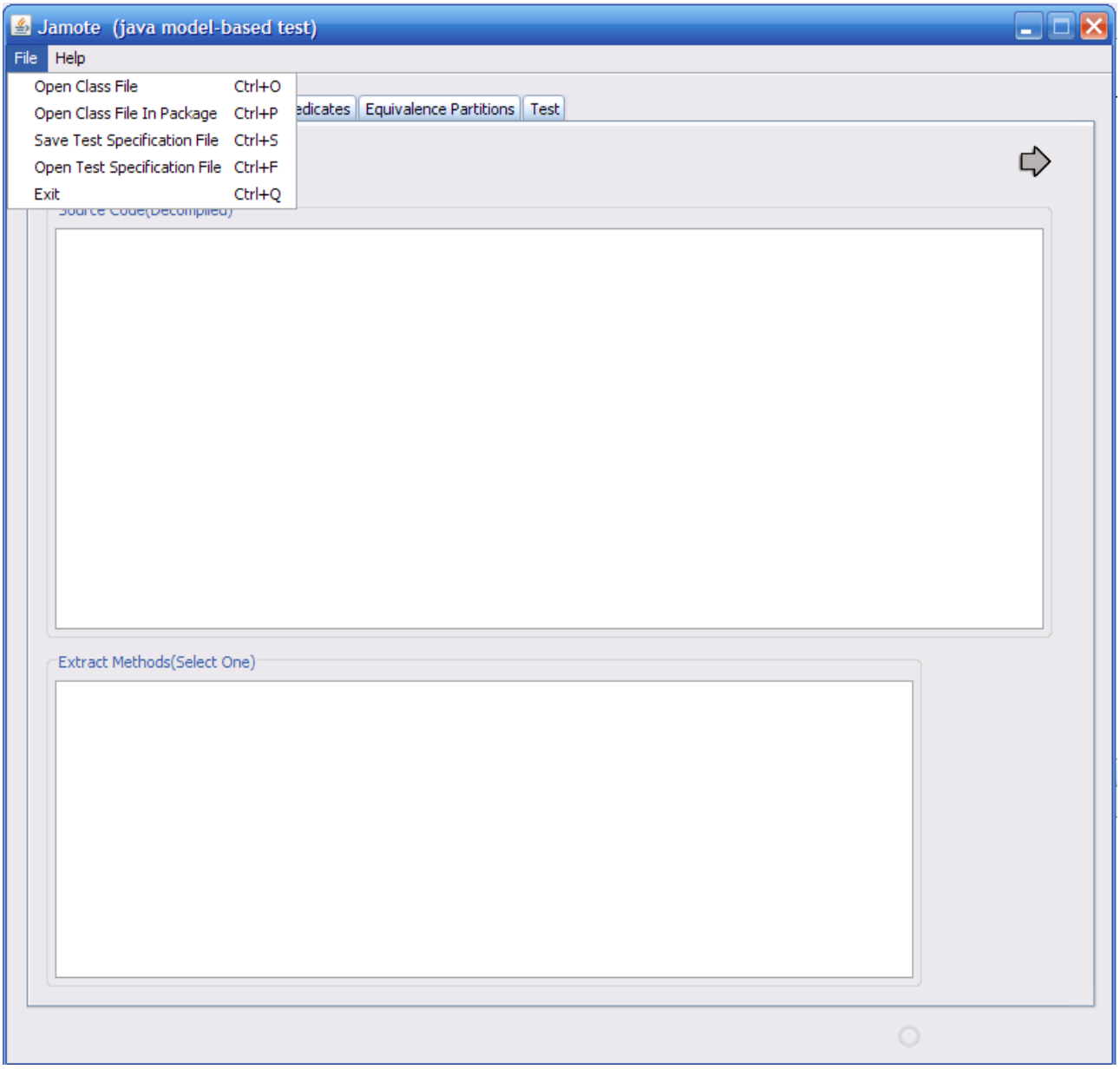


Figura 1: descrizione menù

Il menù File presenta cinque voci:

- Open Class File: permette di caricare un file java compilato .

- Open Class File In A Package: permette di caricare un file java compilato che è inserito in un package.
- Save Test Specification File: consente di salvare in un file xml tutte le precondizioni, le partizioni di equivalenza e le funzioni in sintassi alloy inserite come specifiche del test.
- Open Test Specification File: permette di caricare un file xml,(precedentemente salvato) contenente le specifiche di test, nell' applicazione.
- Exit: chiude l'applicazione.

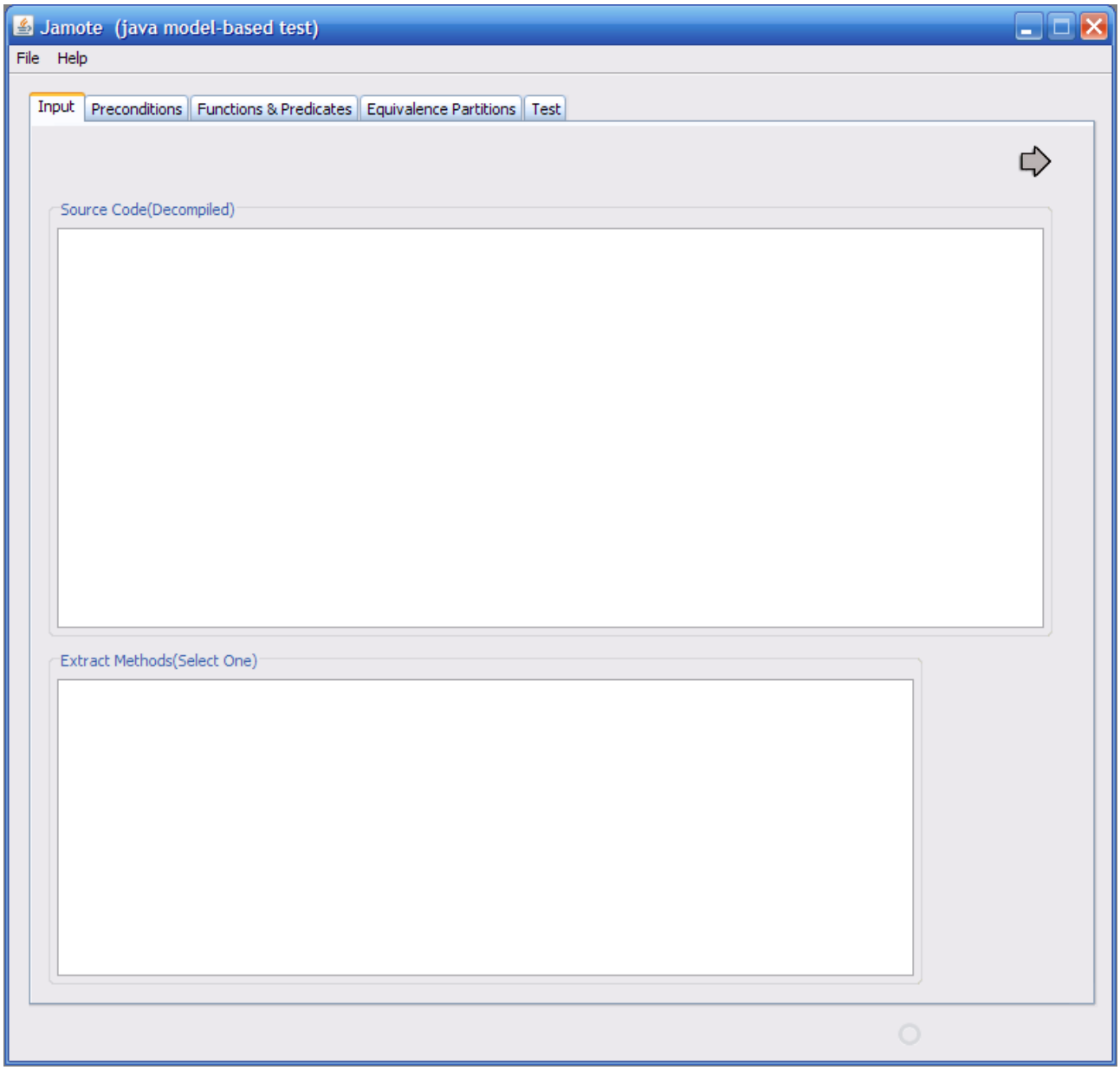


Figura 2: tab Input

Il tab Input contiene due pannelli:

- Source Code: pannello in cui viene visualizzato il codice sorgente decompilato del file .class caricato
- Extract Methods: lista in cui vengono presentati tutti i metodi contenuti nel file .class

caricato e da dove deve essere selezionata la funzione da testare.

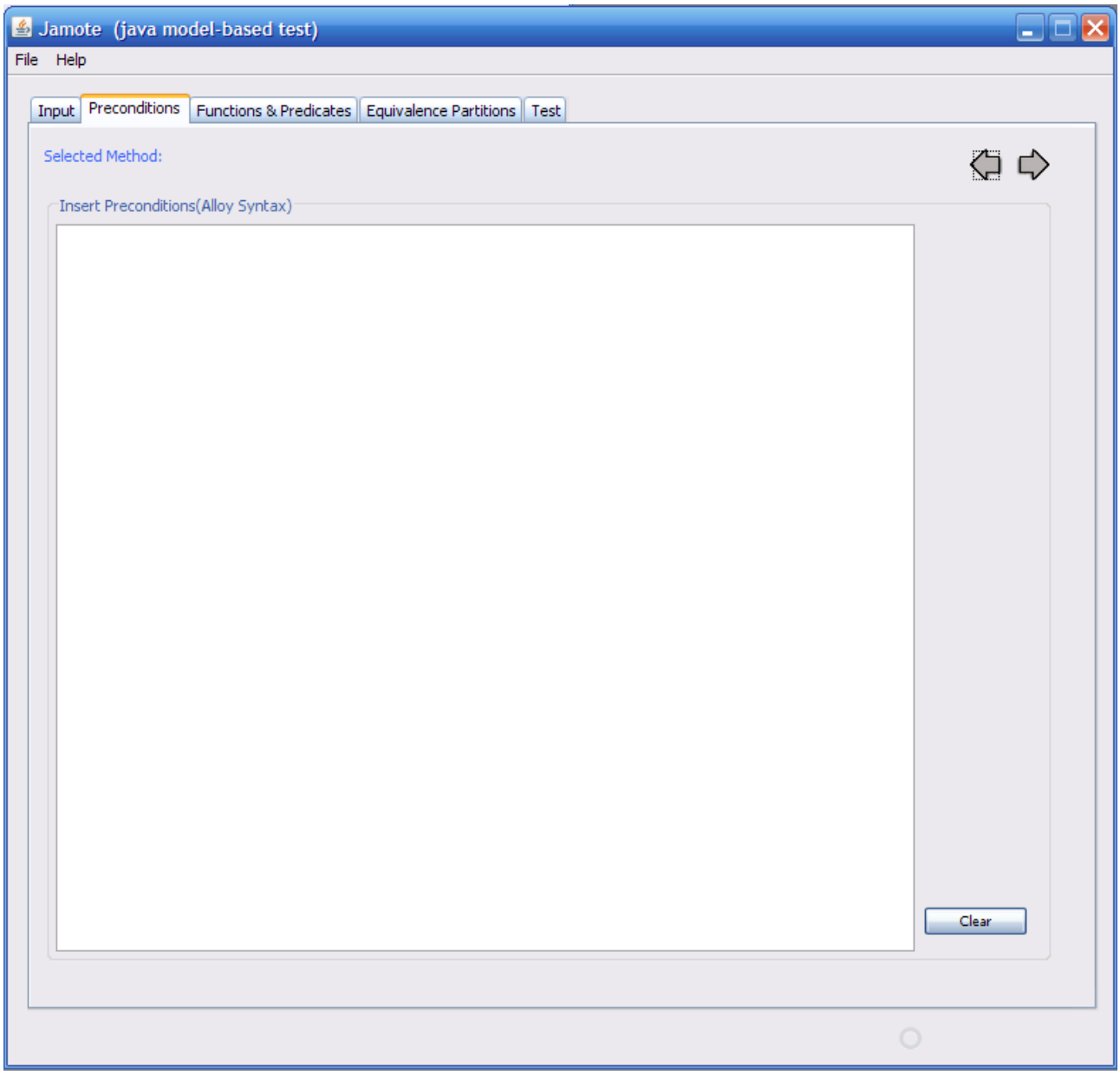


Figura 3: tab Preconditions

Il tab Preconditions contiene un unico pannello:

- Insert Preconditions: qui vanno inserite tutte le precondizioni, ovvero tutti gli input che la funzione da testare non è in grado di gestire. Il tasto clear ha la funzione di cancellare le precondizioni già inserite, qualora non fosse più necessario utilizzarle.

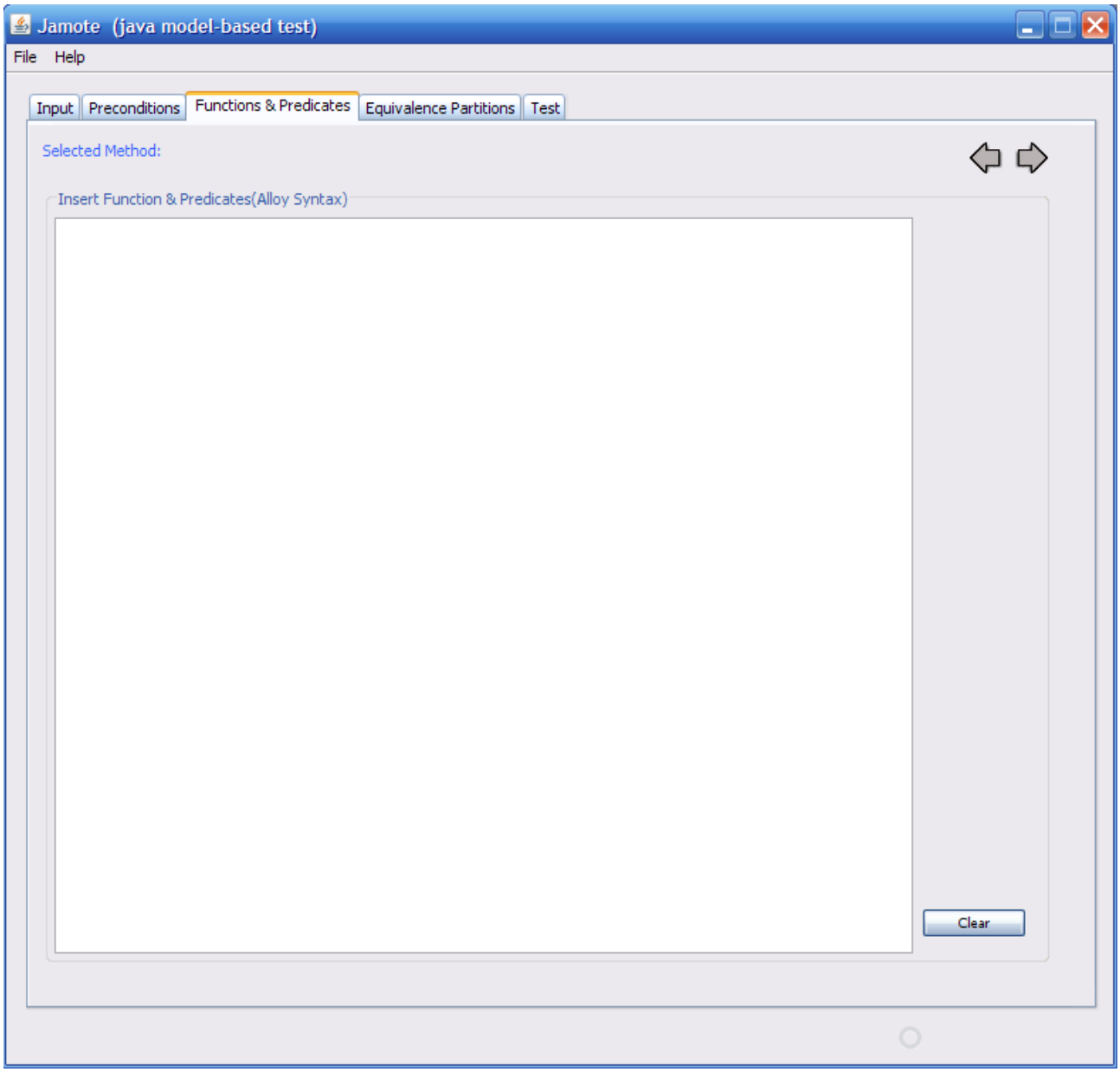


Figura 4: tab Functions & Predicates

Il tab Functions & Predicates è simile al precedente e contiene anch'esso un unico pannello:

- Insert Functions & Predicates: qui vanno inserite funzioni e predicati di supporto che possono servire ad esprimere formule complesse in Alloy come ad esempio la definizione di somma degli elementi di un array, oppure la definizione di ordinamento degli elementi dello stesso. Il tasto clear serve ad eliminare funzioni e predicati già inseriti e non più necessari.

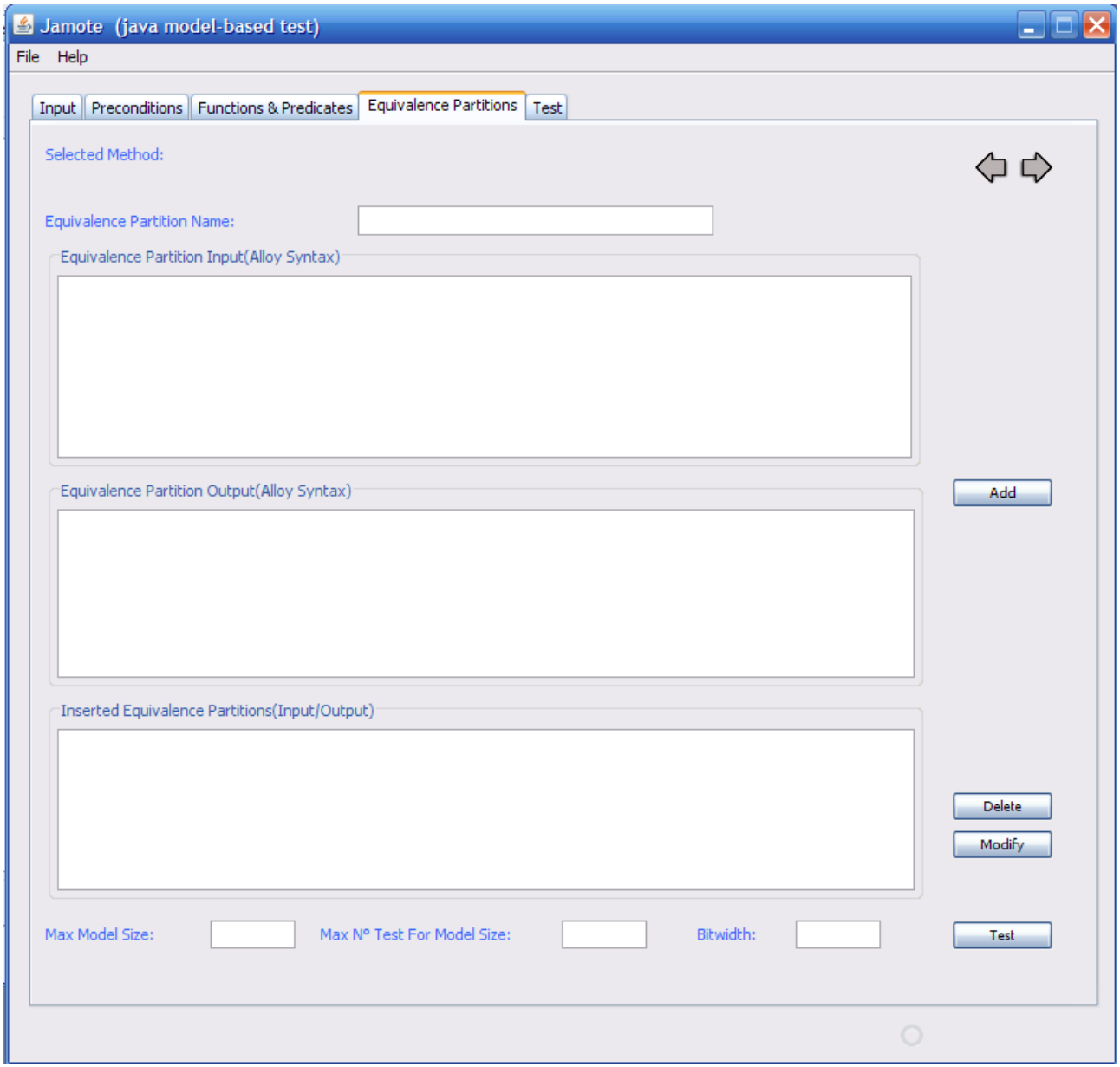


Figura 5: tab Equivalence Partitions

Il tab Equivalence Partitions contiene:

- Equivalence Partition Name: campo di testo in cui può essere inserito il nome della partizione d'equivalenza. Il campo non è obbligatorio.
- Equivalence Partition Input: pannello in cui va inserita la specifica di input della partizione di equivalenza.
- Equivalence Partition Output: pannello in cui va inserita la specifica di output per la partizione di equivalenza.
- Max Model Size: campo di testo in cui inserire la dimensione massima del modello da generare per una determinata partizione d'equivalenza.
- Max Number Of Test: campo di testo che specifica il numero massimo di test per una data partizione d'equivalenza.
- Bitwidth: il numero dei bit utilizzati per rappresentare gli interi all'interno del solver.

- Add button: pulsante che premuto, una volta inseriti tutti i campi precedenti, aggiunge effettivamente la partizione d’equivalenza alle partizioni che verranno prese in considerazione nel test.
- Inserted Equivalence Partitions: lista che contiene tutte le partizioni di equivalenza inserite con i relativi input, output, dimensione massima del modello che Alloy deve trovare, numero massimo di test per partizione d’equivalenza, e numero di bit per rappresentare gli interi in input alla funzione da testare.
- Delete button: pulsante che, selezionata una partizione dalla lista delle partizioni d’equivalenza inserite,elimina definitivamente la partizione d’equivalenza.
- Modify button: pulsante che, una volta selezionata una partizione d’equivalenza, reinserisce tutti i valori della partizione stessa nei vari campi e ne permette la modifica, se necessario.
- Test button: pulsante che lancia l’esecuzione del test a scatola nera sulla funzione selezionata all’inizio, limitatamente a tutte le precondizioni, le funzioni, i predicati e le classi di equivalenza inserite.

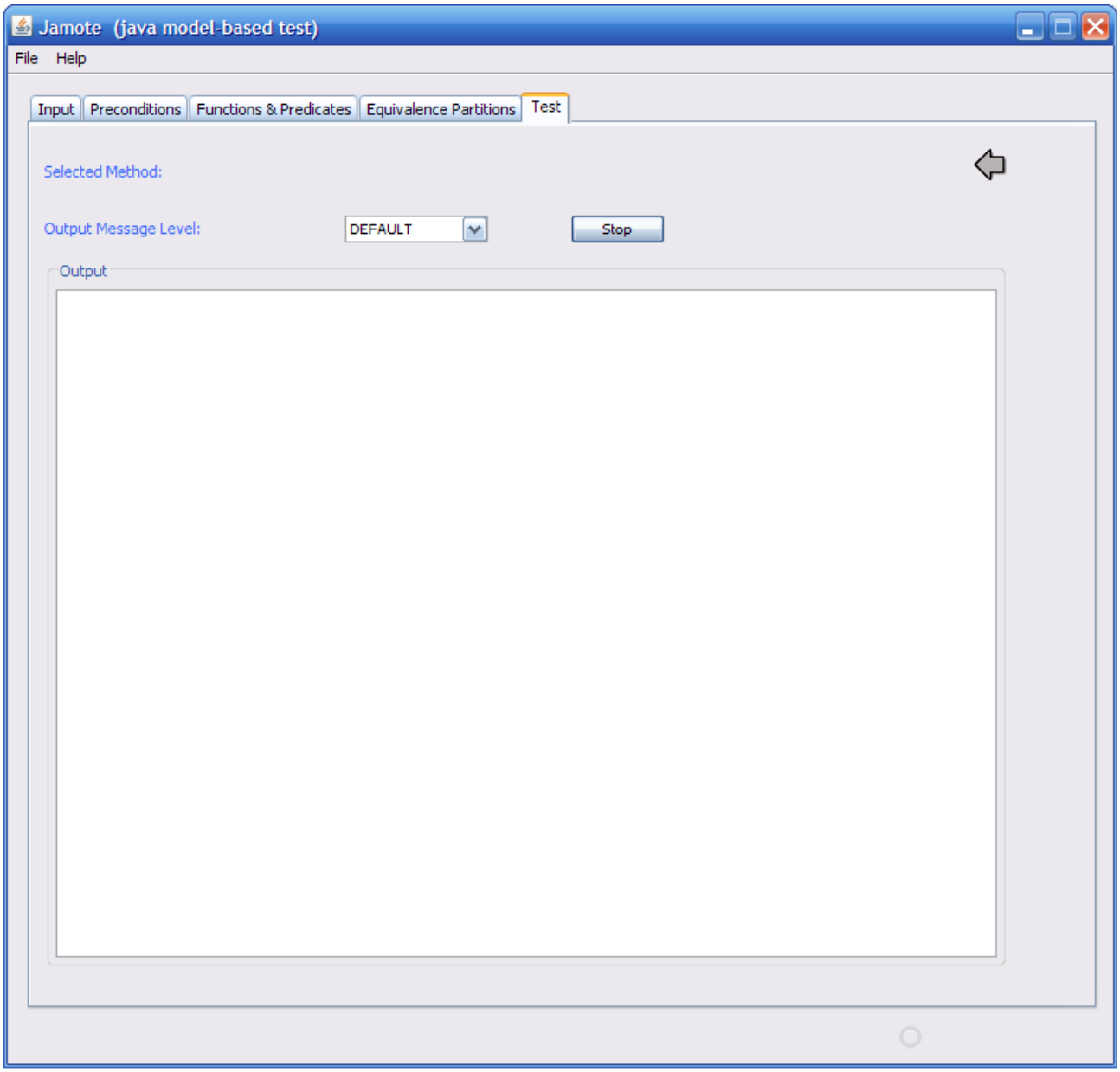


Figura 6: tab Test

Il tab Test contiene:

- Output panel: pannello in cui vengono mostrati tutti i messaggi di output che vengono generati dall'applicazione durante il test della funzione.
- Output Message Level combo box: menù a tendina in cui è possibile selezionare il livello dei messaggi di output che vengono visualizzati nel pannello di output. Le possibili scelte sono:
    - DEBUG : messaggi utili al programmatore per avere una visione dettagliata delle operazioni svolte dall'applicazione in ogni passo dell'esecuzione del test.
    - VERBOSE : messaggi che descrivono tutte le operazioni compiute dall'applicazione con un dettaglio minore dei messaggi di tipo DEBUG.
    - DEFAULT : messaggi che descrivono solo le operazioni, compiute dall'applicazione, necessarie alla comprensione del test da parte dell'utente.

  - WARNING: messaggi che descrivono situazioni di warning generate durante l’esecuzione del test. Ad esempio un test fallito, una situazione di integer overflow, un warning lanciato da Alloy.
  - ERROR: messaggi che rappresentano situazioni di errore, come eccezioni lanciate dal programma o errori lanciati da Alloy come ad esempio errori di parsing.

I messaggi sono elencati in ordine di livello decrescente, una volta selezionato un livello nel menù a tendina, vengono visualizzati tutti i messaggi relativi a quel livello e tutti i messaggi di livello inferiore (se si seleziona DEFAULT vengono stampati tutti i messaggi di tipo DEFAULT ,WARNING ed ERROR).

- Stop button: pulsante che serve a fermare l’esecuzione del test.

4. Esempi di specifiche di test

Nella cartella “assets” sono stati definiti tre esempi di specifiche di test da caricare nell’applicazione in formato XML:

- exampleArraySum
- exampleOrderedArray
- exampleOrderedArrayExc

[4.1] Somma degli elementi di un array

Il file *exampleArraySum.xml* contiene le specifiche per testare una funzione che esegue la somma degli elementi di un array. Le specifiche sono le seguenti:

- **User defined functions & predicates:** Prima di tutto è necessario definire la funzione che calcola la somma degli elementi di un array. E' utile far notare, che Alloy non contiene statements per eseguire loop né tanto meno offre la possibilità di definire funzioni ricorsive. L'unica possibilità che abbiamo per definire funzioni complesse è quella di creare una sig ad hoc che andrà a contenere tutte le combinazioni dei possibili input e i relativi output della funzione. Successivamente si andrà a specificare una serie di vincoli, anche per induzione, su tale sig, che nella pratica andrà a definire l'insieme di tutte le istanze della relazione Input->Output. Tale tecnica è trattata nella pubblicazione [KhuJac00]. In questo caso viene definita la sig *FunArraySum* che utilizza tre valori, l’array a, l’indice b, e l’intero che rappresenta la somma finale c. Il valore del risultato c, viene definito ricorsivamente come la somma di tutti i valori fino all'indice b-1 più il valore all'indice b. La funzione *arraySum* non fa altro che estrarre dalla sig FunArraySum il risultato corrispondente ad uno specifico array e indice array.length – 1.

```
one sig FunArraySum {
	funResult: JamoteArray -> Int -> Int
}
{
	(all a: JamoteArray, b: Int, c: Int | (a->b->c) in funResult
	<=>
	(b >= -1) && (b < a.length) && (
	((b = 0) && c=a.at[b]) ||
	((a.length = 0) && (c=0 && b=-1)) ||
	(some c1:Int | (a->(b-1)->c1 in funResult) && c=add[c1,a.at[b]])
	)) &&
	//Completezza: FunResult deve mappare tutti gli indici di tutti gli array
	(all si: seq/Int,i:Int | some a:JamoteArray, r:Int | (si->i in a.array) => (a->si->r)
in funResult )
}

fun arraySum[a: JamoteArray]: Int
{
	(a.length – 1).(a.(FunArraySum.funResult))
}
```

- **Equivalence partitions**: La partizione di equivalenza è costituita dalla specifica di input e dalla specifica di output. Viene richiesto che in input venga dato un array contenente valori compresi tra la minima e la massima dimensione degli interi, mentre per l'output si richiede che il valore di ritorno sia un intero pari alla somma degli elementi dell'array di input (a tal proposito si utilizza la funzione *arraySum* definita prima). Sempre per l'output viene imposto che non ci siano eccezioni.

```
// Input
(all i:Int | (input.arg0.at[i] <= metadata.maxInt && input.arg0.at[i] >= metadata.minInt ))
&& (all a: JamoteArray | a = input.arg0)

// Output
output.return = arraySum[input.arg0]
&& (one u:univ | output.return = u)
&& (all u:univ | no e:Exception | u = e)
```

[4.2] Ordinamento di un array

Il file Alloy *exampleOrderedArray.xml*, definisce le specifiche di una funzione che ordina gli elementi di un array in maniera non decrescente. Il codice è illustrato qui di seguito:

- **User defined functions & predicates:** Anche in questo caso è necessario definire una funzione addizionale, quella che calcola il numero di occorrenze di un intero all'interno di un array. Anche qui viene definita una sig FunArrayOccurrences, su cui viene applicato il vincolo ricorsivo che impone che il valore di ritorno debba essere uguale al numero di occorrenze di *find* fino all'indice idx – 1 , più 1 nel caso il valore all'indice idx sia uguale a find. La funzione *arrayOccurrences* prende come argomenti l'array e un intero find e estrae da FunArrayOccurrences il numero di le occorrenze di find.

```
one sig FunArrayOccurrences {
	funResult: JamoteArray -> Int -> Int -> Int
}
{
(all a: JamoteArray, idx: Int, find: Int, res:Int | (a->idx->find->res) in funResult
<=>
(idx >= -1) && (idx < a.length) && (
((idx = 0) && (a.at[idx]=find => res=1 else res=0)) ||
((a.length = 0) && (res=0 && idx=-1)) ||
(some res2:Int | (a->(idx-1)->find->res2 in funResult) && (a.at[idx]=find => res=add[1,res2]
else res=res2))
))
//Completezza: FunResult deve mappare tutti gli indici di tutti gli array
(all si:seq/Int, i:Int|some a:JamoteArray, i2:Int | ((si->i in a.array) || si = -1) => (a-
>si->i->i2) in funResult)
}

fun arrayOccurrences[a: JamoteArray, find: Int]: Int
{
	find.((a.length - 1).(a.(FunArrayOccurrences.funResult)))
}
```

- Equivalence Partitions: nella partizione di equivalenza l'input è dato da un qualsiasi array contenente valori compresi tra il massimo e il minimo intero supportato, mentre per l'output viene imposto che l'array di ritorno sia ordinato in maniera non decrescente, ed inoltre il numero di occorrenze di un qualsiasi valore all'interno dell'array di input deve essere uguale al numero di occorrenze di quel valore all'interno dell'array di output, e viceversa. Anche in questo caso, per l'output viene imposto che non ci siano eccezioni.

```
// Input
all i:Int | (input.arg0.at[i] <= metadata.maxInt && input.arg0.at[i] >= metadata.minInt)

// Output
(all idx: seq/Int | (output.return.hasIndex[idx] && output.return.hasIndex[idx+1])=>
output.return.at[idx] <= output.return.at[idx+1])  &&
 (all idx: seq/Int | arrayOccurrences[input.arg0, input.arg0.at[idx]] =
arrayOccurrences[output.return, input.arg0.at[idx]])
  && (all idx: seq/Int | arrayOccurrences[input.arg0, output.return.at[idx]] =
arrayOccurrences[output.return, output.return.at[idx]])
&& (one u:univ | output.return = u)
&& (all u:univ | no e:Exception | u = e)
```

[4.3] Gestione delle eccezioni: ordinamento di un array senza duplicati

Infine il file *exampleOrderedArrayExc.xml* definisce le specifiche di una funzione che prende in input un array non contenente valori duplicati e ritorna il corrispondente array ordinato. Se invece l'array contiene duplicati allora la funzione dovrebbe lanciare un'eccezione.

Il codice è il seguente:

- **User defined functions & predicates:** Uguale al file di esempio analizzato prima.

```
one sig FunArrayOccurrences {
	funResult: JamoteArray -> Int -> Int -> Int
}
{
(all a: JamoteArray, idx: Int, find: Int, res:Int | (a->idx->find->res) in funResult
<=>
(idx >= -1) && (idx < a.length) && (
((idx = 0) && (a.at[idx]=find => res=1 else res=0)) ||
((a.length = 0) && (res=0 && idx=-1)) ||
(some res2:Int | (a->(idx-1)->find->res2 in funResult) && (a.at[idx]=find => res=add[1,res2]
else res=res2))
))
//Completezza: FunResult deve mappare tutti gli indici di tutti gli array
(all si:seq/Int, i:Int|some a:JamoteArray, i2:Int | ((si->i in a.array) || si = -1) => (a-
>si->i->i2) in funResult)
}

fun arrayOccurrences[a: JamoteArray, find: Int]: Int
{
	find.((a.length - 1).(a.(FunArrayOccurrences.funResult)))
}
```

- **Equivalence Partitions (1) – Input con duplicati:** Viene imposto che con input

contenente duplicati la funzione deve lanciare un eccezione di tipo Exception e il valore di ritorno deve essere vuoto.

```
// Input
((all i:Int | input.arg0.at[i] <= metadata.maxInt && input.arg0.at[i] >= metadata.minInt) &&
(some i:Int | input.arg0.arrayOccurrences[input.arg0.at[i]] > 1)

// Output
(no u:univ | output.return = u) && (one e:Exception | output.exception = e)
```

- **Equivalence Partitions (2) – Input senza duplicati:** Viene imposto che con input senza duplicati la funzione ritorni l'array di input ordinato, e non vengano lanciate eccezioni.

```
// Input
all i:Int | (input.arg0.at[i] <= metadata.maxInt && input.arg0.at[i] >= metadata.minInt) &&
input.arg0.arrayOccurrences[input.arg0.at[i]] <= 1

// Output
(all idx: seq/Int | (output.return.hasIndex[idx] && output.return.hasIndex[idx+1])=>
output.return.at[idx] <= output.return.at[idx+1])
&& (one u:univ | output.return = u)
&& (all u:univ | no e:Exception | u = e)
```

[4.4] Le funzioni di test

Infine, le funzioni Java su cui utilizzare tali specifiche di esempio si trovano tutte nel file ***Jamote/TestExample.class*** in particolare:

- **public int** arraySumCorrect(**int**[] a) : esegue correttamente la somma degli elementi dell'array. Utilizzare il file *exampleArraySum.xml* per eseguire il test che dovrebbe terminare con successo.
- **int**[] arrayOrderingCorrect(**int**[] d) : esegue correttamente l'ordinamento di un array con e senza duplicati. Utilizzare il file *exampleOrderedArray.xml* per eseguire il test.
- **int**[] arrayOrderingWithoutDups(**int**[] d) : esegue correttamente l'ordinamento di un array senza duplicati, senza eseguire nessun controllo preventivo sull'input. Utilizzare il file *exampleOrderedArray.xml* per eseguire il test. In questo caso le specifiche non impongono nessuna precondizione sull'input, quindi è possibile che la funzione venga testata con array contenenti duplicati e quindi restituisca un valore non corretto. Si potrebbe anche utilizzare il file *exampleOrderedArrayExc.xml* ma anche in questo caso risulterebbe corretta solo una classe di equivalenza, quella che specifica l'input senza duplicati, mentre la classe di equivalenza che specifica input con duplicati e eccezione lanciata risulterebbe scorretta, poiché tale funzione Java non lancia nessuna eccezione in caso di array con duplicati.

- **int**[] arrayOrderingWithoutDupsExc(**int**[] d) **throws** Exception : questa funzione esegue correttamente l'ordinamento di un array senza duplicati, mentre in caso le venga passato un array contenente duplicati lancia un'eccezione di tipo Exception. Si può utilizzare le specifiche nel file *exampleOrderedArrayExc.xml* per eseguire il test, che dovrebbe terminare senza output scorretti.

[4.5] Analisi delle prestazioni

E' utile, a questo punto, condurre un'analisi sulle prestazioni del sistema. Prima di tutto, bisogna notare che Alloy Analyzer internamente utilizza SAT come “motore”. In generale si può affermare dunque, che il tempo del processo di generazione cresce velocemente in funzione della dimensione del modello stesso. Si noti inoltre che in Alloy ogni numero intero viene modellato come istanza di una sig e quindi eseguire la generazione di un modello con interi a 5 bit, equivale a generare un modello contenente almeno 64 oggetti. Risulta dunque abbastanza arduo eseguire test anche con interi di media dimensione, empiricamente si è verificato che su una macchina con 2Gb di ram la generazione di semplici modelli con interi di dimensione superiore a 10 bit è praticamente impossibile poiché il processo termina con un messaggio di “memoria non sufficiente”. La dimensione degli interi risulta dunque essere il fattore che incide maggiormente sulle prestazioni.

Un altro fattore importante da tenere in considerazione è il numero di relazioni tra gli oggetti. Un esempio classico in cui c'è ”un'esplosione” del numero di tuple di oggetti è quello in cui vengono definite delle sig con relazioni di arità n. E' facile verificare che il numero di tuple che istanziano tali relazioni cresce molto velocemente al crescere dell'arità. Questo si verifica ad esempio in quelle sig che vengono utilizzate per modellare funzioni (come la somma dei valori di un array e il conteggio delle occorrenze di un valore in un array, già analizzate negli esempi) in cui si hanno relazioni del tipo Input->Output. In questo caso, dunque, anche la definizione di costrutti con relazioni complesse risulta incidere in modo significativo sulle prestazioni del sistema.

Il numero di modelli generati risulta essere un fattore che impatta in modo poco significativo sulle prestazioni generali del sistema. A quanto sembra Alloy, una volta trovata la prima soluzione impiega tempi irrisori per trovare le successive. Il discorso, purtroppo, cambia nel caso in cui i modelli trovati contengano istanze isolate. In questo caso, infatti, si rischia di generare modelli isomorfi che vengono scartati automaticamente da Jamote. Se il numero delle possibili istanze isolate cresce, verranno generati molti modelli duplicati e quindi Jamote impiegherà più tempo a riconoscerli e scartarli.

Tanto per farsi un'idea, riportiamo qui un serie di statistiche sulle prestazioni del tool Jamote. Le prove sono state condotte su un laptop con sistema operativo Linux con 2gb di ram, processore Intel Core 2 Duo @1.6 Ghz. I test mettono in evidenza l'impatto della dimensione degli interi e delle sig di supporto alle funzioni complesse, sulle prestazioni del sistema.

Input	***Model size***	***N models***	***Int Bitwidth***	***Tempo***
Array senza vincoli, nessuna sig di supporto	5	1	2	0.407 s
Array senza vincoli, nessuna sig di supporto	5	1	5	0.746 s
Array senza vincoli, nessuna sig di supporto	5	1	8	3.374 s
Array con valori limitati, sig per il calcolo della somma (funzione con arità 2)	5	1	2	0.398 s
Array con valori limitati, sig per il calcolo della somma (funzione con arità 2)	5	1	3	4.795 s
Array con valori limitati, sig per il calcolo della somma (funzione con arità 2)	5	1	4	15.347 s
Array con valori limitati, sig per il calcolo delle occorrenze (funzione con arità 3)	5	1	2	0.834 s
Array con valori limitati, sig per il calcolo delle occorrenze (funzione con arità 3)	5	1	3	22.968 s
Array con valori limitati, sig per il calcolo delle occorrenze (funzione con arità 3)	5	1	4	1 h 8 min

5. Bibliografia

- [KhuMar04] Sarfraz Khurshid and Darko Marinov - *TestEra: Specification-based Testing of Java Programs Using SAT*
- [KhuJac00] Sarfraz Khurshid and Daniel Jackson - *Exploring the Design of an Intentional Naming Scheme with an Automatic Constraint Analyzer*
- [Jac06] Daniel Jackson - *Software Abstractions - Logic, Language, and Analysis*
- *Alloy API Javadoc [http://alloy.mit.edu/alloy4/public/]*
- *Alloy Docs and tutorials @ [http://alloy.mit.edu/alloy4/]*