ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

DEPLOYMENT AUTOMATICO DI APPLICAZIONI SPECIFICATE IN LINGUAGGIO ABS

Relazione finale in
TECNOLOGIE WEB

Relatore
Prof. GIANLUIGI ZAVATTARO

Co-relatore
Prof. JACOPO MAURO

Presentata da
CRISTIAN PAOLUCCI

Seconda Sessione di Laurea
Anno Accademico 2014 – 2015

PAROLE CHIAVE

MODDE
ABS
Zephyrus
Metis
Docker

"Push your business technology into the cloud and get back to focusing on your core competencies"

- Tom Cochran

Indice

Sommario **ix**

Introduzione **xi**

1 ABS **1**
1.1 Descrizione 1
1.2 Architettura 2

2 MODDE **5**
2.1 Descrizione 5
2.2 ABS 6
2.3 DDLang 9
2.4 Architettura 12

3 Zephyrus **15**
3.1 Descrizione 15
3.2 Input 17
3.3 Output 18

4 Metis **21**
4.1 Descrizione 21
4.2 Analisi Algoritmo 22

5 Integrazione MODDE **25**
5.1 Docker 25
5.2 EasyInterface 27
5.3 Installazione 27
5.4 Sviluppi Futuri 35

Conclusioni **37**

Ringraziamenti **39**

Bibliografia **41**

Elenco delle figure **43**

Sommario

Seguendo un percorso suddivisibile essenzialmente in due macro-fasi, la tesi si sviluppa attraverso cinque capitoli. La prima fase, dedicata all'approfondimento di tematiche di ricerca ed alla ricognizione sullo stato dell'arte, è descritta attraverso i primi quattro capitoli. L'ultimo, invece, descrive la successiva fase relativa all'integrazione dello strumento descritto.

Capitolo 1 Il primo capitolo affronta brevemente il linguaggio di specifica utilizzato. In particolare, viene illustrata la sua funzionalità e architettura di base che verrà utilizzata nel capitolo successivo.

Capitolo 2 Il secondo capitolo descrive la struttura principale dello strumento e dei suoi componenti. Più nel dettaglio si descrive l'input, l'output e come vengono utilizzati i linguaggi ABS e DDLang per indicare tutte le specifiche che deve avere il sistema da installare

Capitolo 3 Nel terzo capitolo si entra nel dettaglio dello strumento Zephyrus, con la spiegazione delle sue funzionalità e una vista sulla sua architettura, partendo dall'input fino all'output.

Capitolo 4 Nel quarto capitolo si descrve l'ultimo strumento che utilizza MODDE, ovvero Metis. Specificatamente verrà introdotto il suo algoritmo e verrà spiegato come esso sceglie l'ordine con cui installare i componenti.

Capitolo 5 Nell'ultimo capitolo si discute dell'integrazione di MODDE in un'interfaccia web tramite un server HTTP e la creazione di un container in Docker

Introduzione

Da quando è iniziata l'era del Cloud Computing molte cose sono cambiate, ora è possibile ottenere un server in tempo reale e usare strumenti automatizzati per installarvi applicazioni. Molti nuovi strumenti sono stati creati, però ognuno di essi mira ad una parte del problema e non svolgono un lavoro completo. Per esempio, uno strumento di gestione della configurazione come Chef può fornire un server cloud permettendo di installare e configurare un'applicazione. Comunque Chef necessita della riconfigurazione dell'equilibratore di carico ogni volta che cambiano i server web. Oppure Amazon Auto Scaling può fornire server e aggiornare gli equilibratori, ma dipende da CloudWatch. Quindi tutti gli strumenti mirano a creare e gestire un'applicazione a livello di cloud che si adatti alla domanda degli utenti per garantire quello che si aspettano senza sprecare risorse monetarie su servizi superflui. In ogni caso sono necessari diversi strumenti e quali di questi integrare per pianificare l'installazione è una scelta decisiva. Per esempio il servizio web di Amazon EC2 cerca di fornire tutti gli strumenti per gestire il cloud computing, però si rischia uno stallo dovuto alla totalità degli strumenti progettati dallo stesso fornitore. Per cercare un'uniformità fra i vari strumenti disponibili sul mercato sono stati recentemente creati degli standard: OASIS CAMP (Cloud Application Management for Platforms) e TOSCA (Topology and Orchestration Specification for Cloud). Essi forniscono le basi per costruire diversi strumenti che collaborino tra di loro attraverso l'uso di un linguaggio arricchito basato su XML: YAML. Uno strumento che si basa su di essi, per esempio, è Apache Brooklyn, il quale serve a modellare, monitorare e gestire applicazioni attraverso dei blueprint che utilizzano YAML.

Normalmente per decidere quante macchine assegnare ad un certo tipo di applicazione si fa affidamento ad un team di utenti esperti. Essi calcolano il piano migliore, ma questo è incline all'errore umano e causa un grande dispendio di tempo. Quindi MODDE si propone come alternativa per il deployment automatico dei servizi su cloud. In un ambiente con diversi fornitori e diverse tecnologie garantisce un approccio standard tramite l'uso del linguaggio ABS. Esso fornisce un livello di astrazione sufficiente per descrivere un sistema senza doverlo implementare. Questo permette di capire quante macchine allocare su

un cloud, grazie alle specifiche fornite dal cliente e l'universo dei componenti creato dal fornitore. Con MODDE si propone un approccio dove l'installazione è una componente principale della fase di sviluppo. La produzione di sistemi software moderni adotta frequentemente l'approccio di produzione continua, secondo il quale c'è un continuum tra lo sviluppo e la fase di installazione. Tuttavia, a livello di modellazione, la combinazione fra di essi è molto lontano dal diventare una pratica comune.

Inoltre verranno introdotti i due sotto-strumenti usati da MODDE: Zephyrus e Metis. Il primo si occupa di trovare i componenti da installare tenendo conto di tutte le loro dipendenze, cercando di ottimizzare il risultato. Il secondo gestisce principalmente l'ordine con cui installarli tenendo conto dei loro stati interni. Con la collaborazione di questi componenti si ottiene un installazione automatica piuttosto efficace.

Il percorso sviluppato attraversa vari argomenti, iniziando dalla struttura del linguaggio proposto fino ai vari strumenti utilizzati nel dettaglio. Una volta esaminati nella loro struttura generale si procede con la spiegazione di una integrazione proposta su un interfaccia web installata su server HTTP Apache.

Capitolo 1

ABS

1.1 Descrizione

Il linguaggio ABS è un linguaggio a oggetti basato su classi che include tipi di dato algebrici e funzioni senza effetti collaterali. Sintatticamente, l'ABS prova ad essere più simile possibile a Java, in modo che i programmatori che lo usano possano facilmente usare ABS senza troppi sforzi.
ABS offre diverse funzionalità come: chiamate dei metodi asincrone, funzioni per controllare queste chamate, interfacce per l'incapsulazione e programmazione cooperativa dei metodi di invocazione dentro gli oggetti attivi. Più nello specifico ogni oggetto creato in ABS rappresenta un attore con dati incapsulati. Similarmente a java il loro comportamento e lo stato sono definiti implementando interfacce con i loro rispettivi metodi. Quindi esse interagiscono eseguendo chiamate asincrone a questi metodi, i quali generano messaggi che vengono poi inseriti in una specifica coda per ogni attore. L'attore continua prendendo ogni messaggio dalla coda e processandolo, eseguendo il metodo al quale corrisponde. Questa combinazione di caratteristiche risulta in un modello a oggetti concorrente che è composto in maniera ereditaria. La semplicità di ABS quindi deriva dal fatto che ogni attore è come un processore separato, il che lo rende adatto per modellare applicazioni distribuite.
ABS quindi prende di mira software concorrenti, distribuiti, a oggetti, composti da diversi componenti e riusabili. Essendo un linguaggio astratto è adatto per modellare software che dovrebbe essere usato in un sistema virtualizzato. Per poterlo effettivamente applicare è comunque necessario rappresentare concetti di basso livello, come: memoria, latenza, tempo e pianificazione. Con ABS è possibile farlo tramite una notazione chiamata deployment components.
ABS quindi non è solo a scopo di modellazione, ma possiede un sistema di strumenti integrati che aiutano ad automatizzare tutti i processi di ingegneria

del software. Purtroppo gli strumenti da soli sono inutili, ammenoché non venga garantita la prevedibilità dei risultati, interoperabilità e la possibilità d'uso. Un requisito fondamentale dei primi due criteri è l'uniformità, tramite una semantica formale. Però l'interoperabiltà include la capacità di connettersi con altre annotazioni che non siano ABS, quindi vengono fornite diverse interfacce di linguaggio.

1.2 Architettura

L'architettura di ABS è organizzata come una pila di strati separati come in figura:

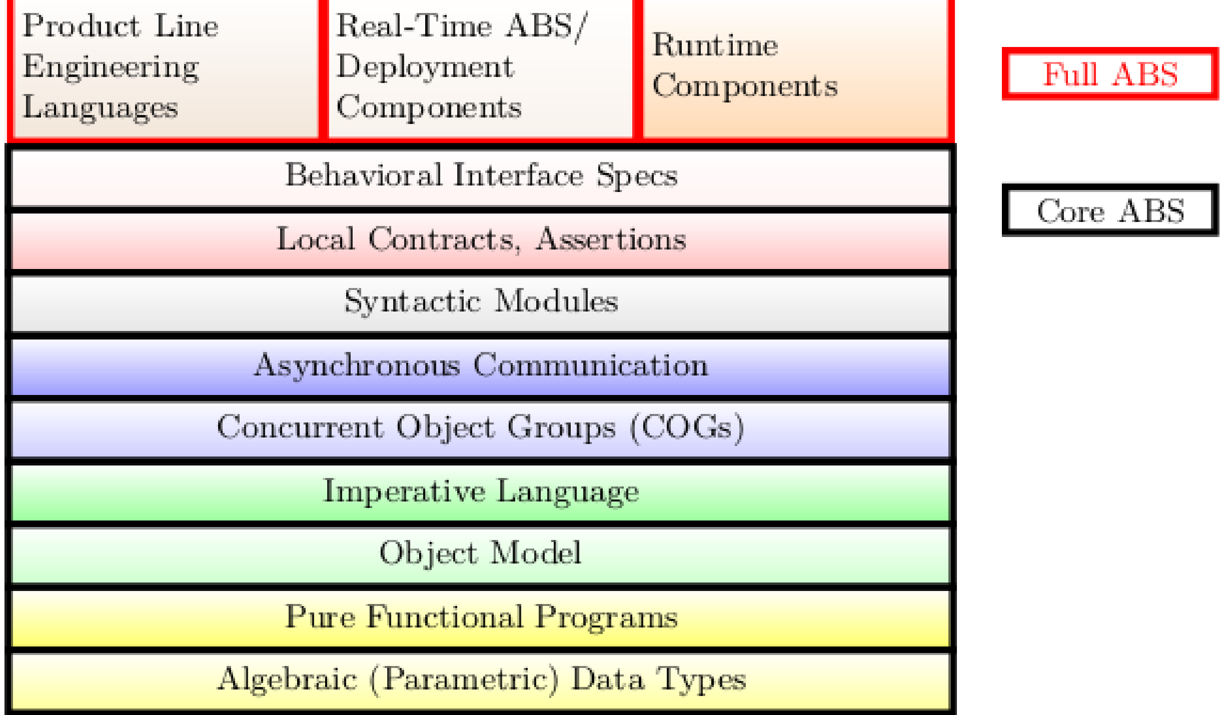


Figura 1.1: Architettura di ABS

I quattro strati inferiori forniscono un linguaggio di programmazione basato su una combinazione di tipi di dato algebrico, funzioni pure e un semplice linguaggio a oggetti imperativo. I due livelli successivi sono molto legati fra di loro in quanto realizzano la concorrenza distribuita.

ABS è un linguaggio completamente eseguibile, tuttavia l'astrazione è raggiunta in diversi modi:

- ABS contiene solo cinque tipi di dato, tutto il resto è definito dall'utente. Il fondamento logico è che nessuna decisione prematura sulle proprietà è forzata, il che aiuta a creare un modello indipendente dall'implementazione.

- Le funzioni sui tipi di dato possono essere non specificati. Il modellatore ha la possibilità di restituire valori astratti o lasciarle incomplete. L'ultimo può causare errori in fase di esecuzione, tuttavia risultano utili per la simulazione, test o scenari di verifica.

- La pianificazione dei compiti, come l'ordine dei messaggi in coda, è non deterministica.

Se ad un certo punto fosse necessario dare i dettagli di un'implementazione completa si potrebbe raffinare un tipo di dato algebrico in una classe implementata. Oppure realizzandola in Java tramite la interfaccia per i linguaggi esterni disponibile in ABS. Le capacità di astrazione di ABS consentono di specificare il comportamento parziale durante il design, senza dover effettivamente dare i dettagli dell'implementazione.

Capitolo 2

MODDE

2.1 Descrizione

Nei sistemi software moderni è sempre più frequente osservare un continuo fra la fase di sviluppo e la fase di installazione. A livello di modellazione questo continuum è lontano dal diventare una pratica comune. Infatti i le tecniche di modellazione tradizionali supportano la fase di sviluppo, mentre linguaggi di modellazione più recenti si incentrano sull'installazione dell'applicazione. L'approccio usato si basa su tre colonne portanti:

- Modellazione degli artefatti del software per comporre il sistema desiderato: la loro descrizione è arricchita con l'indicazione delle loro dipendenze funzionali e la quantificazione delle risorse necessarie per poter essere eseguite

- Linguaggio di alto livello per la specificazione della distribuzione desiderata: i requisiti minimi possono essere espressi per distribuire il sistema come i componenti basici che devono essere presenti o il numero di repliche di un servizio.

- Un sistema automatizzato che, presi in ingresso sia i requisiti degli artefatti software che l'aspettativa globale, crei una distribuzione che soddisfi entrambi i vincoli e possibilmente ottimizzi alcuni obiettivi.

Sono previsti diversi vantaggi dall'uso di aspetti legati alla distribuzione in fase di modellazione. Questi permettono di effettuare analisi di differenti alternative di distribuzione nelle prime fasi fornendo al team di sviluppo un supporto sulle decisioni. È possibile inoltre notare la necessità di aggiungere più iterazioni nel design nel caso in cui l'analisi non sia soddisfacente. In questo caso non è necessario effettuare test su installazioni reali per verificare se una scelta ha un impatto negativo sul sistema.

Il linguaggio di modellazione usato è ABS. Esso, come è stato spiegato nel precedente capitolo, supporta la modellazione di sistemi distribuiti rappresentati come una rete di componenti di distribuzione. Questi ultimi sono contenitori che forniscono oggetti che comunicano concorrentemente in modo asincrono.

Gli argomenti principali discussi saranno:

- Le estensioni di ABS con la possibilità di annotare le definizioni delle classi con informazioni di installazione. Diversi scenari possono essere considerati e, per ognuno di essi, è possibile indicare le funzionalità e i requisiti di risorse per gli oggetti di quelle classi.
- La definizione del DDLang, un linguaggio specifico che permette di dichiarare le specifiche di alto livello desiderate.
- L'implementazione di un motore di distribuzione basato sul modello (MODDE), uno strumento che, preso un set di classi ABS e le specifiche in DDLang, crei un programma principale ABS che allochi gli oggetti desiderati. I componenti da allocare sono presi da una descrizione delle risorse date a MODDE come un file addizionale in formato JSON.

È necessario dire che per l'implementazione di MODDE vengono usati due strumenti già esistenti: Zephyrus e Metis. Il primo supporta il calcolo dell'allocazione ottimale degli oggetti dati i componenti, mentre il secondo supporta la generazione della sequenza con cui eseguire le azioni nel programma principale.

2.2 ABS

Il linguaggio ABS è stato creato per sviluppare modelli eseguibili con uno stile di programmazione orientato a oggetti. ABS si incentra su sistemi distribuiti concorrenti. Inoltre ABS supporta diverse tecniche per l'analisi della modellazione basate su una semantica formale. Gli elementi di base sono i componenti da allocare.

```
DeploymentComponent small = new DeploymentComponent("m1",
   map[Pair(Memory, 500), Pair(CPU, 1)]);
DeploymentComponent large = new DeploymentComponent("m2",
   map[Pair(Memory, 1500), Pair(CPU, 4)]);
[DC: large] Service s1 = new Service();
[DC: large] Service s2 = new Service();
[DC: small] Balancer b = new Balancer(list[s1, s2]);
```

Listato 2.1: Deployment Component

Nel codice ABS qui sopra vengono creati due componenti: small e large. Ogni componente ha una stringa di identificazione e un set di risorse. Vengono creati tre oggetti: i primi due sono servizi posizionati nei componenti large, mentre il terzo è un equilibratore localizzato su un componente small. In ABS è possibile dichiarare gerarchie fra interfacce e definire le classi che le implementano.

```
interface EndPoint {}
interface ReverseProxy extends EndPoint {}
class Balancer(List<Service> services)
  implements ReverseProxy { ... }
```

Listato 2.2: Gerarchie delle Interfacce

Nel pezzo di codice ReverseProxy è dichiarato come interfaccia che estende EndPoint e la classe Balancer è definita come una sua implementazione. I parametri di inizializzazione richiesti durante l'istanziazione degli oggetti sono indicati come parametri nella definizione delle classi. Come detto sopra l'inizializzazione dei parametri della classe Balancer consiste in una lista di servizi da bilanciare. Per la corretta generazione del programma principale è necessaria un'annotazione per ogni classe rilevante coinvolta. Per esempio non è necessario annotare una classe che implementa una struttura di dati interna. Una annotazione per una classe C descrive:

- Il massimo consumo di risorse di un oggetto Obj della classe C
- I requisiti dei parametri di inizializzazione della classe C (per esempio: almeno due servizi dovrebbero essere presenti nella lista di un equilibratore di carico)
- Quanti oggetti possono usare le funzionalità di Obj nel sistema

```
ann
  : '[ Deploy: scenario [' expr (',' expr )* ']] ';
expr
  : 'Name (' STRING ')'
  | 'MaxUse (' INT ')'
  | 'Cost (' STRING ',' INT ')'
  | 'Param (' STRING ',' paramKind ')';
paramKind
  : User
  | 'Default (' STRING ')'
  | Req
  | 'List (' INT ')';
```

Listato 2.3: Grammatica delle annotazioni ABS

Qui sopra si può vedere la grammatica ANTLR del linguaggio di annotazione. Generalmente una annotazione è semplicemente una lista di espressioni separate da virgole, dove le espressioni sono dei seguenti tipi:

- Name(X): associa un nome X all'annotazione. Il nome la identifica inequivocabilmente in caso ne siano più di una, ognuna rappresenta un modo differente di creare oggetti di quella classe. Questa espressione può essere non specificata in una delle annotazioni al massimo, in questo caso ne viene assegnata una di default.

- MaxUse(X): indica che un oggetto della classe può essere usato nella creazione di al massimo X altri oggetti. Questo parametro esprime il vincolo che nello scenario specificato l'oggetto può dare funzionalità ad un numero limitato di altri oggetti. Se questo campo è vuoto possono essere creati oggetti illimitati.

- Cost(r, X): Indica che un oggetto della classe consuma al massimo X unità della risorsa r.

- Param(param, kind): indica come i parametri di inizializzazione param della classe devono essere istanziati quando si crea un oggetto della classe. Ce ne sono quattro casi:

  - User: l'utente deve inserire il nome del parametro. Questo avviene quando l'utente è l'unico che sa come specificare il parametro. In questo caso va istanziato manualmente.
  - Default(X): il parametro è impostato al valore di default X.
  - Req: il parametro necessita di essere definito da MODDE, il quale è responsabile di creare un oggetto appropriato e passarlo come parametro quando l'oggetto è istanziato.
  - List(X): il parametro richiede una lista di almeno X oggetti che dovrebbero essere definiti da MODDE.

```
interface IQueryService extends Service {
   List < Item > doQuery ( String q);
}
[ Deploy : scenario [
   MaxUse (1) ,
   Cost (" CPU " , 1) , Cost (" Memory " , 400) ,
```

```
   Param ("c" , Default (" CustomerX ") ,
   Param (" ds " , Req ) ]]
class QueryServiceImpl ( DeploymentService ds ,
   Customer c) implements IQueryService {
      // Implementazione
}
```

Listato 2.4: Esempio ABS

Nell'esempio sopra è possibile vedere un esempio di codice ABS. Astraendo l'implementazione si può vedere la classe QueryServiceImpl che implementa l'interfaccia IQueryService. L'annotazione per la classe QueryServiceImpl è introdotta prima della definizione della classe. L'annotazione MaxUse(1) specifica che un oggetto di QueryServiceImpl potrebbe essere usato come parametro solo una volta durante la creazione di altri oggetti. Con Cost invece associa un costo a un oggetto di QueryServiceImpl. In questo caso un oggetto di quella classe può consumare fino a 4GB di memoria e una CPU. Con Param si annotano i singoli parametri di inizializzazione della classe. QueryServiceImpl ha due parametri: ds, il quale è un oggetto che implementa l'interfaccia di DeploymentService, e il cliente c. ds è impostato come parametro richiesto. Questo significa che prima di creare un oggetto di QueryServiceImpl è necessario creare un oggetto che implementa DeploymentService. Il parametro cliente, invece, è impostato a default, ovvero CustomerX.

2.3 DDLang

Quando un sistema è stato generato automaticamente un utente si aspetta di raggiungere degli obiettivi. Questi possono essere espressi tramite il Linguaggio di distribuzione dichiarativa (DDLang): un linguaggio per definire i vincoli che la configurazione finale dovrebbe soddisfare.

```
spec
   : expr comparisonOP expr |
   | spec boolOP spec | 'true ' |
   | 'not ' spec | '(' spec ')' ;
expr
   : 'DC[' resourceFilter '|' simpleExpr ']'
   | 'DC[' simpleExpr ']'
   | expr arithmeticOP expr
   | simpleExpr ;
resourceFilter
   : STRING comparisonOP INT
   | resourceFilter ';' resourceFilter ;
```

```
simpleExpr
   : exprNoDC comparisonOP exprNoDC
   | simpleExpr boolOP simpleExpr |
   | 'true ' | 'not ' spec | '(' spec ')' ;
exprNoDC :
   INT |
   'INTERFACE [' STRING ']'|
   'CLASS [' STRING ']' |
   'CLASS [' STRING ':' STRING ']' |
   exprNoDC arithmeticOP exprNoDC ;
comparisonOP : ' <=' | '<' | '=' | ' >=' | '>' ;
arithmeticOP : '+' | '-' | '*' ;
boolOP : 'and ' | 'or ' | 'impl ' | 'iff ' ;
```

Listato 2.5: Grammatica DDLang

Come mostrato nel codice qui sopra un vincolo in DDLang è una specificazione di vincoli base expr comparisonOP expr (riga 2) combinati usando i collegamenti logici classici. Questi vincoli base indicano quanti elementi l'utente desidera creare. Una espressione può identificare diversi tipi di quantità base:

- Un intero
- Il numero di oggetti che implementano l'interfaccia
- Il numero di oggetti della classe

In quest'ultimo caso è possibile indicare il numero di oggetti della classe creati secondo lo scenario S. Con questa espressività è possibile aggiungere vincoli che si astraggono dai componenti. Per esempio potrebbe essere necessario creare due oggetti che implementano l'interfaccia IQueryService ed esattamente 1 oggetto della classe PlatformServiceImpl usando l'espressione:

```
INTERFACE[IQueryService] >= 2 and
CLASS[PlatformServiceImpl] = 1
```

Quantità più complesse riguardano i componenti. Essi sono espressi alla riga 6 con la notazione DC[filter — simpleExpr] dove filter è una sequenza di vincoli sulle risorse fornite dal componente e simpleExpr è una espressione. DC[filter — simpleExpr] denota il numero di componenti che soddisfano i vincoli sulle risorse di filter e che contengono oggetti che soddisfano l'espressione di simpleExpr. Per esempio è possibile specificare che nessun componente che abbia meno di due CPU contenga più di un oggetto della classe QueryServiceImpl.

```
DC[ CPU <= 2 | CLASS[QueryServiceImpl] >= 2 ] = 0
```

Usando questi vincoli è possibile esprimere co-allocazioni o richieste di distribuzione. Per motivi di efficienza potrebbe essere conveniente co-allocare oggetti che hanno molte interazioni fra di loro. Per esempio consideriamo il caso nel quale sia necessario co-installare con un oggetto QueryServiceImpl con uno DeploymentServiceImpl.

```
DC[ CLASS[QueryServiceImpl] > 0 and
CLASS[DeploymentServiceImpl] = 0 ] = 0
```

L'impossibilità di co-installare due oggetti invece si può esprimere come:

```
DC[ CLASS[PlatformServiceImpl] > 0 and
CLASS[LoadBalancerServiceImpl] > 0 ] = 0
```

2.4 Architettura

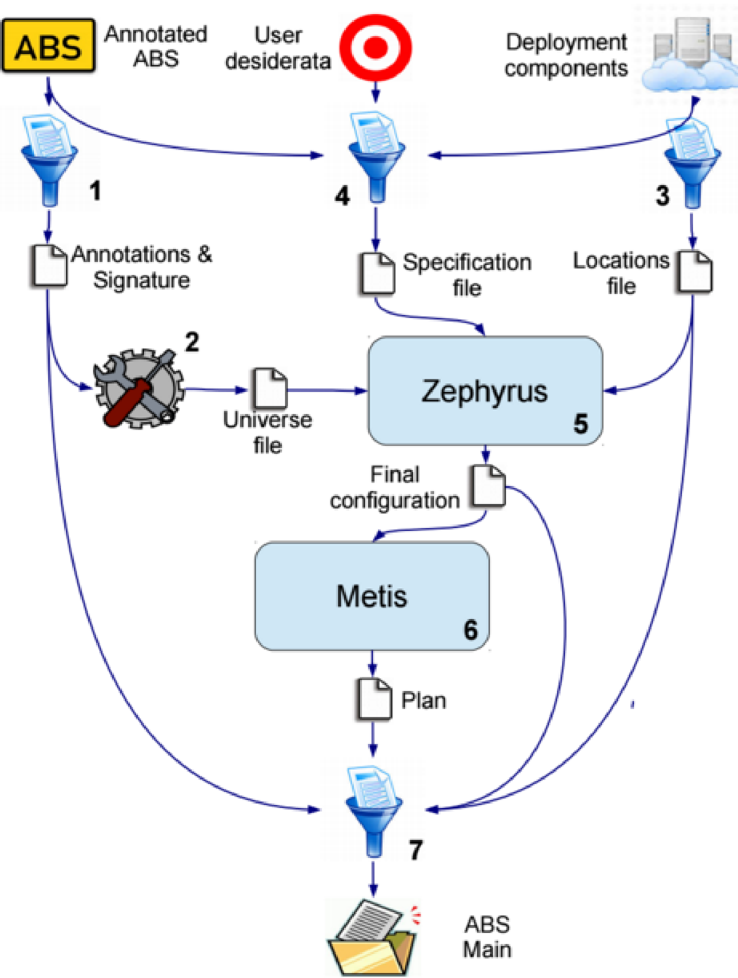


Figura 2.1: Flusso di esecuzione di MODDE

MODDE è lo strumento usato per generare un programma principale ABS che crea gli oggetti ottenuti tramite istanziazione delle classi annotate che soddisfano i vincoli espressi in DDLang. MODDE utilizza degli script che integrano Zephyrus e Metis seguendo lo schema nella figura. Più precisamente prende tre input:

- Un programma ABS
- Le preferenze dell'utente formalizzate come vincoli espressi in DDLang

- La lista dei componenti disponibili, espressi come descritti sotto

I componenti vengono dati come oggetti JSON con due proprietà: DC_description, il quale descrive i differenti tipi di componenti e DC_availability che specifica il numero di istanze disponibili. Un tipo di componente è definito da un nome, la lista delle risorse che fornisce e un costo (monetario) che l'utente dovrà pagare per usarlo.

```
{
    " DC_description ": [
   { " name " : "c3. large ",
      " provide_resources " :
         {"CPU" : 2, " Memory " : 375} ,
       " cost " : 105 } ,
    { " name " : "c3. xlarge ",
      " provide_resources " :
         {"CPU" : 4, " Memory " : 750} ,
      " cost " : 210 } ],
    " DC_availability ": {
      "c3. large " : 5,
      "c3. xlarge " : 3 }
}
```

Per esempio il componente c3.large fornisce 2 CPU e 3.75 GB di RAM e se usata costa 105 crediti all'ora.

Quando MODDE viene eseguito viene costruito un albero di programmi ABS, dal quale si ricavano tutte le annotazioni e le classi. Questo passaggio (punto 1) è fatto da un programma Java che restituisce un file JSON. Nel passaggio successivo l'output è processato per generare l'universo dei componenti richiesto da Zephyrus. Come verrà spiegato più in dettaglio dopo nel capitolo su Zephyrus, esso richiede come input una rappresentazione dei componenti da distribuire seguendo le specifiche del modello Aeolus. In Aeolus ogni componente è una scatola chiusa che mostra i propri stati interni durante il processo di creazione. Ogni stato gestisce le porte che rappresentano le funzionalità di un componente. Le porte richiedenti devono essere legate alle porte fornitrici di altri componenti. In Aeolus è possibile associare numeri alla porta per gestire i vincoli di capacità o replicazione. Per le porte richiedenti questo valore indica il numero minimo di componenti distinti che soddisfano i requisiti. Invece per le porte fornitrici indica il massimo numero di componenti che possono usare le funzionalità. Zephyrus necessita di altri due input: descrizione di tutte le allocazioni e i requisiti delle configurazioni finali. Questi due input vengono calcolati nei passaggi 3 e 4. Nel punto 4 i vincoli in DDLang vengono tradotti nel linguaggio di specifica di Zephyrus. Una volta raccolti

tutti gli input viene eseguito Zephyrus. Per generare il programma principale in ABS rimane soltanto da stabilire l'ordine in cui installare i vari oggetti. Per ottenere questa informazione viene lanciato Metis (punto 6). Esso prende in input la configurazione finale prodotta da Zephyrus e l'universo ottenuto al punto 2 e ne deriva le azioni da svolgere per ottenere la configurazione finale. Dopo aver ottenuto la pianificazione da Metis è possibile procedere e generare il programma ABS. I componenti da usare sono stati calcolati da Zephyrus. Successivamente, basandosi sull'ordine calcolato da Metis, i nuovi oggetti sono creati e allocati nei corrispondenti componenti.

Capitolo 3

Zephyrus

3.1 Descrizione

In contrasto ai classici software monolitici che funzionano localmente su una singola macchina, i sistemi largamente distribuiti sono costituiti da molti servizi su macchine virtuali che collaborano per provvedere le funzionalità aspettate dall'utente finale. Gli architetti del sistema devono:

- Scegliere quali servizi usare e come configurarli, sapendo che essi potrebbero avere delle dipendenze o conflitti tra di loro
- Considerare la tolleranza agli errori e garantire abbastanza istanze per ogni servizio
- Creare l'architettura fisica sulla quale installare il sistema, provando a tenere bassi i costi ma con il giusto numero di allocazioni con abbastanza risorse in modo da permettere una buona esecuzione dei servizi che ospitano
- Scegliere quale implementazione di ogni servizio installare su quale allocazione, sapendo che le implementazioni (spesso in forma di packages) hanno a loro volta dipendenze e conflitti.

Una volta pianificato tutto questo, la fase di distribuzione deve fornire le macchine virtuali richieste, installare i giusti package su ognuna di esse e infine iniziare a collegare tutti i servizi nel giusto ordine.

Questo è un compito scoraggiante, come per costruire un puzzle, ogni servizio, package o macchina, sono tasselli, dove si conosce solo la funzionalità aspettata nell'intero. Per ridurre la complessità di questo processo, molte iniziative a livello industriale sviluppano strumenti che permettono di selezionare,

configurare e caricare su Cloud dei servizi ben definiti. Comunque questi strumenti sono utili solo una volta che il puzzle è stato risolto, per esempio quando sono stati scelti i servizi adeguati, le allocazioni sulle quali vanno distribuiti e una maniera per configurarli secondo i requisiti. Risolvere questo puzzle per ora richiede un significante dispendio di risorse umane, infatti nella pratica vengono sempre gestite con degli script personalizzati o manualmente. Questo metodo è particolarmente incline agli errori. Qui viene presentata una catena di strumenti sviluppata nella struttura del progetto Aeolus che fornisce una automazione generica e alternativa alle tecniche elencate precedentemente. Essa si compone di due strumenti:

Zephyrus, che genera automaticamente una rappresentazione astratta del sistema designato secondo una concisa specificazione delle funzionalità aspettate. Esso considera la totalità dei servizi disponibili, i quali potrebbero servire come blocchi da costruzione con i loro:

- Requisiti
- Politiche di replicazione
- Caratteristiche di consumo delle risorse
- Informazioni riguardo l'implementazione dei servizi
- Massimo numero di macchine (virtuali) disponibili, assieme alle loro caratteristiche.

Zephyrus è anche capace di minimizzare l'ammontare di risorse necessarie.

Armonic, prende le configurazioni di sistema prodotte da Zephyrus e le alloca fornendo il numero di macchine virtuali su una piattaforma di calcolo Cloud, installando i package necessari su ogni macchina. Esso configura i servizi per stabilire le connessioni richieste e infine li avvia nell'ordine giusto, basandosi su dei precisi metadati che descrivono lo stato interno e le dipendenze di ogni servizio in fase di esecuzione.

Questa catena di strumenti distacca il design dall'installazione del sistema e si costruisce sopra le fondamenta del modello di Aeolus. Esso descrive ogni servizio come componente, usando delle porte con un tipologia che include i requisiti, necessità, politiche di replicazione e una macchina a stati interna per cogliere il ciclo di vita dei componenti. Zephyrus usa una versione del modello Aeolus senza stati, estesa per tener conto delle allocazioni, repositories, package e risorse. Le specifiche accettate da Zephyrus sono date in una sintassi rigorosa le cui semantiche definiscono quando la configurazione soddisfa una specifica. Basandosi su queste formalizzazioni Zephyrus può essere definito corretto e completo: troverà sempre una configurazione ottimale per quanto riguarda

il criterio scelto, se esiste. Inoltre è garantito che la configurazione generata fornisca le funzionalità aspettate e soddisfi i vincoli definiti dalle politiche di replicazione oltre alle dipendenze e conflitti fra servizi. Armonic quindi usa le informazioni della macchina a stati interna dei suoi componenti per determinare una corretta sequenza di attivazione dei servizi, assumendo che le relazioni di dipendenza dei servizi siano acicliche. Questo non lo rende uno strumento ottimale per MODDE in quanto le relazioni possono essere cicliche. Nel prossimo capitolo verrà appunto introdotto Metis.

3.2 Input

Zephyrus prende tre input:

- Una descrizione di tutti i componenti e i loro vincoli, i quali sono forniti in vari formati per via delle differenti origini; questo è chiamato universo.
- Una descrizione della configurazione corrente del sistema (quali servizi sono installati e dove, macchine esistenti e altro)
- Una specificazione di alto livello del sistema desiderato. Come parte della specifica, gli architetti possono includere funzioni obiettivo per le quali bisogna ottimizzare, come la minimizzazione del numero di macchine virtuali che verranno usate per la distribuzione.

Zephyrus quindi prende come input una descrizione dei tipi di servizio disponibili e una relazione di implementazione che mappa ogni servizio per il set di package che lo implementano. Queste due parti dell'universo vengono date a Zephyrus come file JSON, come nell'esempio:

```
{ "component_types": [
   { "name" : "DNS-load-balancer",
      "provide" : [["@wordpress-frontend"], ["@dns"]],
      "require" : [["@wordpress-backend", 7]],
      "conflict": ["@dns"],
      "consume" : [["ram", 128]] },
   { "name" : "HTTP-load-balancer",
      "provide" : [["@wordpress-frontend"]],
      "require" : [["@wordpress-backend", 3]],
      "consume" : [["ram", 2048]] },
{ "name" : "Wordpress",
"provide" : [["@wordpress-backend"]],
"require" : [["@sql", 2]],
"consume" : [["ram", 512]] },
```

```
{ "name" : "MySQL",
"provide" : [["@sql", 3]],
"consume" : [["ram", 512]] } ],
"implementation": [
[ "DNS-load-balancer", ["bind9"] ],
[ "HTTP-load-balancer", ["varnish"] ],
[ "Wordpress", ["wordpress"] ],
[ "MySQL", ["mysql-server"] ] ] }
```

La sezione component_types descrive i tipi di component disponibili con le loro porte insieme ai requisiti non funzionali, come memoria o banda. A differenza di altri strumenti Zephyrus ha piena coscienza dei package disponibili con le loro dipendenze e conflitti e assicura che a livello di package sia tutto rispettato. È possibile quindi associare repositories con package differenti a varie allocazioni per coadiuvare l'installazione su sistemi eterogenei. Un'altra parte importante dell'input di Zephyrus è quella di descrivere la configurazione iniziale, come i tipi di macchine disponibili con le proprie risorse:

```
{ "locations" : [
{ "name" : "loc1",
"repository" : "debian-squeeze",
"provide_resources" : [["ram", 2048]] },
{ "name" : "loc2",
"repository" : "debian-squeeze",
"provide_resources" : [["ram", 2048]] },
[...] }
```

Attualmente Zephyrus offre due ottimizzazioni di base chiamate compatta e conservativa. La prima mira a minimizzare il numero di componenti usati, mentre la seconda cerca di ridurre le differenze basandosi sulla configurazione iniziale.

3.3 Output

L'output di Zephyrus contiene una descrizione completa del sistema da distribuire

```
{ "locations": [
{ "name": "loc1",
"provide_resources": [ [ "ram", 2048 ] ],
"repository": "debian-squeeze",
"packages_installed": [ "wordpress" ] },
{ "name": "loc2",
```

```
"provide_resources": [ [ "ram", 2048 ] ],
"repository": "debian-squeeze",
"packages_installed": ["mysql-server",
"wordpress" ] } [...]
```


Ogni allocazione è associata ad una lista di package che dovrebbero essere installati. Solo i root package sono elencati e Zephyrus ha già controllato la loro compatibilità, soddisfando le dipendenze e i conflitti. La seconda parte dell'output è la lista delle istanze dei servizi, mappate sulle loro allocazioni:


```
"components": [
{ "name": "Wordpress-1",
"type": "Wordpress",
"location": "loc1" },
{ "name": "Wordpress-2",
"type": "Wordpress",
"location": "loc2" },
{ "name": "MySQL-1", "type": "MySQL",
"location": "loc2" }, [...]
```


La terza parte elenca i collegamenti che connettono i servizi fra loro:


```
"bindings": [
{ "port": "@wordpress-backend",
"requirer": "HTTP-load-balancer-1",
"provider": "Wordpress-1" },
{ "port": "@wordpress-backend",
"requirer": "HTTP-load-balancer-1",
"provider": "Wordpress-2" },
{ "port": "@sql",
"requirer": "Wordpress-1",
"provider": "MySQL-1" } [...]
```


Partendo dall'output di Zephyrus, Armonic crea un numero di macchine virtuali sufficiente per ogni istanza e con le informazioni sulle risorse fornite si scelgono quelle di dimensione adeguata. Zephyrus può dover trattare migliaia di package per ogni locazione. Per facilitare l'operazione esso esegue diversi passi di semplificazione sull'input che ne riduce notevolmente la mole: l'universo viene rifilato rimuovendo tutti i servizi che non sono nella chiusura transitiva degli altri servizi presenti nella configurazione iniziale o nella richiesta. I package che implementano i servizi rimossi vengono a loro volta eliminati. Successivamente Zephyrus traduce l'input ridotto in un set di vincoli su numeri interi non negativi. Questi vincoli usano variabili differenti per ogni

istanza per creare un allocazione per ognuno dei tipi nell'universo. I vincoli impongono che le istanze rispettino le definizioni del loro tipo nell'universo, il modo in cui sono implementate nei package, i conflitti fra di loro, le dipendenze e le specificazioni del problema. L'abilità di catturare come vincoli interi l'esistenza di un'intera architettura corrispondente alle specifiche è il punto di forza di questo approccio. Questo ci permette di gestire le diverse sfaccettature del design del sistema nella sua totalità e quindi garantire la completezza dello strumento e la correttezza della configurazione generata. I vincoli che sono stati generati vengono espressi tramite il linguaggio di modellazione dei vincoli MiniZinc. Questo permette di usare un qualsiasi risolutore di vincoli compatibile con esso. Una volta risolti i vincoli Zephyrus trasforma la soluzione, la quale è una mappatura di variabili in valori interi, in una configurazione vera e propria. Nella generazione bisogna riuscire a riusare più parti esistenti possibili e creare collegamenti fra istanze nel modo più corretto, tenendo conto che due istanze possono essere possono essere legate ad una porta solo una volta. Così abbiamo introdotto un approccio automatizzato per creare e distribuire complesse applicazioni composte da servizi interconnessi, come spesso si trovano nei moderni ambienti Cloud. L'architettura del sistema può: specificare le componenti necessarie per ottenere le funzionalità richieste, aggiungere vincoli non funzionali, risorse disponibili e dichiarare incompatibilità fra componenti. Il maggior vantaggio dell'approccio proposto è che tutti i vincoli esistenti sono presi in considerazione per prevenire errori al momento dell'installazione.

Capitolo 4

Metis

4.1 Descrizione

Collocare dei sistemi di componenti software sta diventando una sfida, specialmente da quando sono nate le tecnologie di Cloud Computing che rendono possibile l'esecuzione rapida di software distribuiti su richiesta e in una infrastruttura virtuale. Quando il numero di componenti software necessari cresce le loro dipendenze diventano troppo complesse per essere gestite manualmente. È importante che l'amministratore usi linguaggi di alto livello per specificare i requisiti minimi del sistema e successivamente si affidi a strumenti che sintetizzano automaticamente le azioni di basso livello. Queste ultime sono necessarie per realizzare una configurazione di sistema completa e corretta che soddisfa i requisiti.

Quello che interessa è il problema della sintesi automatica del piano di allocazione in presenza di dipendenze circolari fra i componenti. Per studiare il problema consideriamo il modello Aeolus, che arricchisce i componenti del modello standard, basandosi su delle porte fornitrici/richiedenti con una macchina a stati interna che ne descrive il ciclo di vita. Ogni stato interno può attivare solo alcune porte nell'interfaccia del componente. Automatizzare un piano di allocazione consiste nella specificazione di una sequenza di azioni di basso livello come creazione/cancellazione dei componenti, collegamento ad una porta e cambiamenti di stati interni, per raggiungere una configurazione con almeno un componente in ogni stato interno desiderato.

Consideriamo una nuova soluzione per l'allocazione automatica dei componenti basata su un algoritmo diviso in tre fasi. Nella prima si stabilisce un piano effettuando un'analisi di raggiungibilità simbolica in anticipo di tutti gli stati raggiungibili dei componenti. Se lo stato è raggiungibile la seconda fase di pianificazione astratta genera un grafo che indica le tipologie di azioni di modifica dello stato interno che sono necessarie e le dipendenze fra di loro. Le

dipendenze causali riflettono il fatto che un componente dovrebbe effettuare un cambiamento di stato e impossessarsi di una porta prima che lo faccia un altro componente. Nella terza fase di generazione del piano viene effettuato un ordinamento topologico adattativo del piano astratto. Con adattativo si intende che il piano astratto potrebbe essere modificato durante l'ordinamento se è necessario duplicare un componente. La duplicazione è usata per far fronte ai casi in cui più istanze dello stesso tipo di componente debbano essere allocate contemporaneamente, con stati diversi, per usare porte differenti allo stesso momento. Per valutare l'effettiva attuabilità dell'approccio proposto è stato sviluppato uno strumento chiamato METIS (Modern Engineered Tool for Installing Software systems) per sintetizzare piani di allocazione.

4.2 Analisi Algoritmo

Analizziamo ora l'algoritmo usato partendo dai tre punti descritti prima

```
deploymentPlanner(U,<Ttarget, qtarget>){
reachabilityAnalysis() // Crea grafo di raggiungibilita' (nodi
    finali in Nodesn)
if <Ttarget, qtarget> is an element of Nodesn then
   componentSelection() // Seleziona un set di nodi sufficienti per
       raggiungere il target
   abstractPlan()  // Genera un piano astratto
   planSynthesis()    // Genera un piano concreto
}
```

Listato 4.1: Algoritmo base di Metis

La procedura deploymentPlanner nell'algoritmo qui sopra calcola se esiste un piano per raggiungere la configurazione con almeno un componente di tipo T_{target} nello stato q_{target} considerando l'universo dei componenti U.

L'algoritmo invoca inizialmente la funzione reachabilityAnalisis(), la quale genera un grafo di raggiungibilità usato per per controllare se il piano esiste. Questo grafo contiene tutte le coppie $\langle T, q\rangle$ per le quali è possibile raggiungere una configurazione con almeno un componente del tipo T nello stato q. I nodi nel grafo sono organizzati in strati $Nodes_0$, $Nodes_1$, ..., $Nodes_n$ che sono generati in fasi susseguenti. Inizialmente $Nodes_0$ contiene tutte le coppie $\langle T, T.init\rangle$ corrispondenti agli stati iniziali. Dato $Nodes_j$, lo strato $Nodes_{j+1}$ è generato copiando tutte le coppie già disponibili in $Nodes_j$ e aggiungendo le nuove coppie che possono essere raggiunte con delle transizioni dagli stati di $Nodes_j$, dando per scontato che siano disponibili nel contesto dei componenti del tipo e stato $\langle T, q\rangle$. L'analisi termina dato che c'è un numero finito di pos-

sibili coppie tipo-stato di componenti. Con $Nodes_n$ si indicano i nodi generati nell'ultima fase. Se la coppia desiderata è in $Nodes_n$ allora esiste almeno un piano e si procede con componentSelection(). Esso seleziona un set di coppie tipo-stato di componenti sufficiente per raggiungere l'obiettivo. L'idea è di procedere a ritroso dalla coppia obiettivo in $Nodes_n$, selezionando per ogni strato $Nodes_j$ un set di coppie sufficienti per raggiungere quelle che sono già state selezionate nello strato $Nodes_{j+1}$. Più precisamente bisogna considerare più aspetti quando si selezionano i nodi in $Nodes_j$. Una coppia$\langle T, q\rangle$ in $Nodes_{j+1}$ può essere ottenuta sia come una copia di una coppia in $Nodes_j$ o eseguendo un cambio di stato da una coppia $\langle T, q'\rangle$ in $Nodes_j$, assumendo che esista una transizione da q' a q. In questo caso è necessario prendere in considerazione le porte richiedenti che vengono attivate dallo stato q. Se ci sono nuovi requisiti, per permettere di eseguire il cambio di stato, potrebbe essere necessario selezionare coppie addizionali in $Nodes_j$. Usando i nodi selezionati nel grafo la prossima funzione abstractPlan() genera il piano astratto. Esso è un grafo i cui nodi rappresentano le azioni da eseguire e gli archi denotano le dipendenze temporali. Ci sono tre tipi di dipendenze:

- Quelle che rappresentano i vincoli di ordinamento sulle transizioni fra stati che devono essere eseguiti in sequenza sullo stesso componente.
- Quelle che indicano che uno stato q in un componente deve essere ottenuto prima di uno stato q' in un altro perché q' richiede le porte fornite da q.
- Quelle che indicano che uno stato q in un componente deve essere lasciato prima di lasciare lo stato q' in un altro componente perché q richiede le porte fornite da q'.

Un piano potrebbe essere generato in principio effettuando una visita topologica dei nodi nel piano astratto anche se questo tipo di visita non è sempre possibile a causa della presenza di cicli. Intuitivamente i cicli rappresentano situazioni in cui uno stato che cambia da q a q' è allo stesso tempo richiesto, per attivare una nuova porta p' fornita da q', e impedito, perché altrimenti una porta p fornita da q diventerebbe inattiva. Il problema si risolve con la funzione planSynthesis che esegue un ordinamento topologico adattivo: il grafo viene modificato quando si incontra un ciclo duplicando il componente richiesto per eseguire il cambio di stato da q a q'. Un'istanza rimane nello stato q continuando a fornire la porta p, mentre l'altra cambia in q' per poter attivare la porta p'.

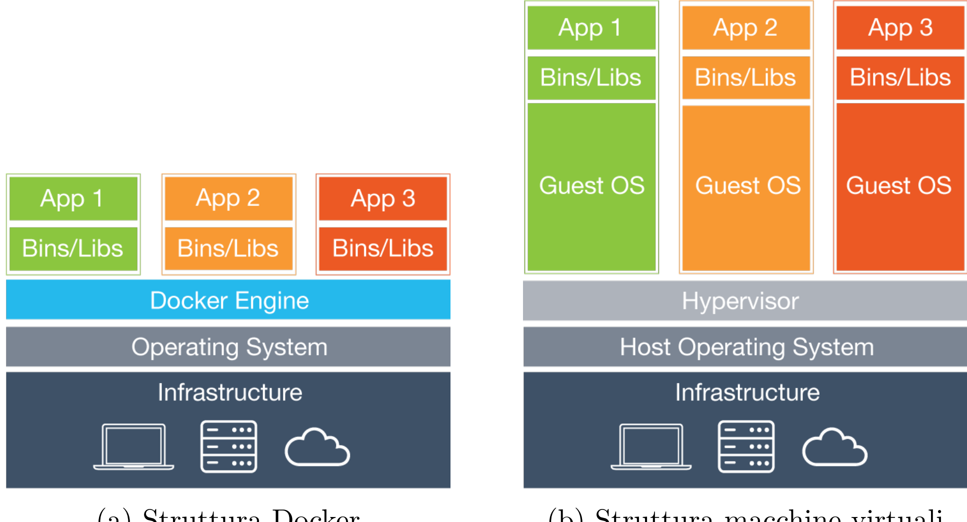
Capitolo 5

Integrazione MODDE

Nonostante MODDE automatizzi tutti i processi per generare il programma principale ABS rimane un software che deve essere eseguito manualmente da un utente competente in quanto necessita della gestione dell'input e richiede un interprete di Python. Un'idea per rendere più semplice l'utilizzo di questo strumento è quello di racchiuderlo in un contenitore nel quale possa essere eseguito senza problemi su qualsiasi sistema operativo, aumentando portabilità e semplicità con un'unica soluzione. Per questo compito è stato usato uno strumento che permette di impacchettare un'applicazione con tutti i suoi requisiti in un'unità standardizzata, ovvero Docker.

5.1 Docker

I contenitori di docker includono frammenti di software su un filesystem completo che contiene tutto il necessario per eseguire qualsiasi applicazione ci sia installata sopra. Questo garantisce che verrà eseguita allo stesso modo indipendentemente dall'ambiente nel quale viene usata. Il vantaggio principale per il quale è stato scelto Docker è la leggerezza: i contenitori condividono tutti lo stesso kernel del sistema operativo, per cui partono istantaneamente e fanno un uso migliore della RAM. Le immagini virtuali sono costruite da filesystem stratificati, quindi possono condividere fra loro eventuali file per diminuire il carico.



(a) Struttura Docker (b) Struttura macchine virtuali

Figura 5.1: Differenze fra Docker e macchine virtuali

I contenitori di docker hanno un isolamento delle risorse e benefici di allocazione simili alle normali macchine virtuali, ma un approccio architetturale diverso. Ogni macchina virtuale include l'applicazione, il necessario per usarla e un intero sistema operativo. I container includono l'applicazione e le dipendenze, ma condividono il kernel con gli altri. Quindi più container possono lavorare assieme alleggerendo ancora di più il sistema.

Una volta creato il container con al suo interno MODDE può semplicemente essere avviato tramite una console con il comando docker run come quello mostrato sotto.

```
docker run -t -v /etc/EIFiles:/abs_deployer/EIFiles modde/tool
    python abs_deployer.py -o EIFiles/out.txt EIFiles/abs.abs
    EIFiles/spec.spec EIFiles/json.json
```

Listato 5.1: Comando per eseguire un'immagine in docker

- -v /etc/EIFiles:/abs_deployer/EIFiles: per passare dei file all'immagine contenente lo strumento è necessario usare dei volumi. Per utilizzarli basta inserire il tag -v, poi si specifica quale cartella del sistema operativo in uso passare e dove collocarla dentro l'immagine (cartella_su_pc:cartella_in_immagine)

- modde/tool: è il nome che è stato associato in questo caso all'immagine di MODDE quando è stata creata

- python abs_deployer.py: serve per avviare il vero e proprio strumento tramite l'interprete di python inserito nell'immagine
- -o EIFiles/out.txt: ridirezione dell'output (ovvero il programma ABS) in un file di testo
- EIFiles/abs.abs EIFiles/spec.spec EIFiles/json.json: sono i tre file di input che richiede MODDE

5.2 EasyInterface

Ora è più semplice utilizzare MODDE, però è comunque necessario aver installato docker sul proprio pc e aver creato la sua l'immagine virtuale. Per migliorare l'integrazione, l'usabilità e la portabilità ad un ulteriore livello si può utilizzare un interfaccia web che consenta di utilizzarlo tramite un server sul quale è installato. Questa interfaccia è la EasyInterface. Essa è una struttura per creare rapidamente una interfaccia per applicazioni in ABS. In pochi minuti è possibile inserire un'applicazione esterna e farla funzionare tramite un'interfaccia web. Tutto quello che serve è un file di configurazione che descrive come avviare l'applicazione. Oltretutto è possibile modificare l'applicazione per utilizzare il linguaggio di output della EasyInterface, il quale è basato su XML, che permette di visionare l'output graficamente. L'EasyInterface è costituita da un lato server, dove viene installata l'applicazione (nel nostro caso Docker, con all'interno l'immagine di MODDE), e di un lato client, il quale permette di creare file nell'editor di testo e di eseguire l'applicazione sul server passando i file scelti. Per installare l'applicazione sul server è necessario creare i file di configurazione normalmente, però docker non può essere eseguito direttamente, perché necessita di una minima gestione dei file in input e in output, quindi è avviato tramite uno script creato appositamente ed è quindi possibile spostare i file in input all'interno della cartella che poi diverrà un volume e mostrare nella console del client l'output in forma testuale.

5.3 Installazione

Ora procediamo ad una breve spiegazione di come viene installato MODDE su un sistema linux. Ovviamente non verranno dati tutti i particolari su come installare tutti i componenti nello specifico per evitare di allungarsi troppo con dettagli che possono essere trovati anche in guide degli sviluppatori esterni nel dettaglio. Innanzitutto è necessario creare un'immagine tramite l'uso di un dockerfile che facilita la costruzione.

```
# install the abs_deployer tool
RUN cd / && \
git clone https://github.com/jacopoMauro/abs_deployer.git

ENV PATH /abs_deployer:$PATH

# copy metis and zephyrus binaries
COPY ./metis.native /bin/metis.native
COPY ./zephyrus.native /bin/zephyrus.native

# install minizinc suite
RUN cd / && \
wget
    http://www.minizinc.org/downloads/release-1.6/minizinc-1.6-x86_64
    -unknown-linux-gnu.tar.gz && \
tar -zxvf minizinc-1.6-x86_64-unknown-linux-gnu.tar.gz && \
rm -rf minizinc-1.6-x86_64-unknown-linux-gnu.tar.gz && \
cd /minizinc-1.6 && \
./SETUP
ENV PATH /minizinc-1.6/bin:$PATH

# install packages
RUN apt-get update && \2  apt-get install -y python
    openjdk-7-jre-headless && \
rm -rf /var/lib/apt/lists/*

WORKDIR /abs_deployer
#ENTRYPOINT ["sunny-cp"]
```

Listato 5.2: Dockerfile

Al suo interno vengono specificati i componenti da includere nell'immagine. Si inizia il processo clonando il repository online di MODDE come nella riga 3. Successivamente si modifica la variabile path dell'immagine (riga 5) e si procede ad aggiungere i componenti di Zephyrus, Metis e minizinc (righe 8, 9 e da 12 a 18), necessari per il funzionamento di MODDE come discusso nei capitoli precedenti. Infine si installa l'interprete di python necessario per eseguire internamente lo strumento. Una volta creato il dockerfile e una cartella contenente tutti i componenti necessari, con il comando "docker build" viene generata l'immagine. Una volta creata l'immagine è necessario installare un supporto sul quale installare la EasyInterface. Su sistemi linux si può semplicemente installare un server HTTP Apache con un modulo per il PHP. Questo può essere semplicemente risolto tramite i comandi:

```
sudo apt-get install apache2
sudo apt-get install php5 libapache2-mod-php5 php5-mcrypt
sudo service apache2 restart
```

Listato 5.3: Installare Apache con PHP5

Ora per inserire la EasyInterface all'interno è semplicemente necessario modificare il file alias.conf (/etc/apache2/mods-enabled/alias.conf) e includere il path della EasyInterface aggiungendo il seguente codice:

```
Alias /ei "/path-to/easyinterface"

<Directory "/path-to/easyinterface">
Options FollowSymlinks MultiViews Indexes IncludesNoExec
AllowOverride All
Require all granted
</Directory>
```

Listato 5.4: Installare Apache con PHP5

Dopo un riavvio del server sarà possibile accedere al client all'indirizzo http://localhost/ei.

Ora verrà descritto come sono i file di configurazione richiesti dalla EasyInterface e come funziona lo script creato per eseguire l'immagine di docker. Prima di tutto è necessario segnalare al server l'intenzione di inserire un nuovo file di configurazione aggiungendo nella lista generale delle applicazioni dove verrà collocato. Questo si ottiene inserendo il comando nel file apps.conf:

```
<app id="dockerApp" src="./default/dockerApp.cfg" />
```

Il path fornito è relativo alla cartella del server in quanto deve essere letto da esso. Ora va creato il file di configurazione dell'applicazione con il nome indicato sopra come segue:

```
<app>
  <appinfo>
    <acronym>DockerApp</acronym>
    <title>DockerApp</title>
  </appinfo>
  <execinfo method="cmdline">
    <cmdlineapp>./default/dockerApp.sh -f _ei_files</cmdlineapp>
  </execinfo>
</app>
```

Gli unici dati non banali sono execinfo e cmdlineapp. Il primo descrive il metodo con il quale il server deve eseguire l'applicazione, ovvero linea di comando in questo caso. Il secondo invece descrive il path e gli eventuali parametri, come -f che indica il passaggio dei file in generale. Questi verranno inviati come una lista all'applicazione. Lo script viene avviato alla pressione del pulsante Apply dell'interfaccia. Esso prende in input una lista dei file, in questo caso 3: un programma abs, un file di specifica e un file JSON nel caso ottimale. Successivamente viene fatto un controllo sulle estensioni per verificare che siano stati passati i file correttamente.

```
. default/parse_params.sh

abs=false
spec=false
json=false

for f in $files
do
   ext=${f##*.}
   if [ $ext = "abs" ]; then
      abs=true
      mv $f "/etc/EIFiles/abs.abs"
   elif [ $ext = "spec" ]; then
      spec=true
      mv $f "/etc/EIFiles/spec.spec"
   elif [ $ext = "json" ]; then
      json=true
      mv $f "/etc/EIFiles/json.json"
   fi;
done

[...]
```

Listato 5.5: Script EI

Se tutti i file sono stati inviati correttamente vengo fatti processare da docker.

```
[...]

if [ "$abs" = true ] && [ "$spec" = true ] && [ "$json" = true ];
    then
```

```
   sudo docker run -t -v /etc/EIFiles:/abs_deployer/EIFiles
      modde/tool python abs_deployer.py -o EIFiles/out.txt
      EIFiles/abs.abs EIFiles/spec.spec EIFiles/json.json
   echo

   cat "/etc/EIFiles/out.txt"

else

[...]
```

Listato 5.6: Script EI

Con il comando cat viene mostrato l'output nella console della EasyInterface. Se un file fosse assente verrebbe mostrato nel client tramite il linguaggio di output dell'interfaccia.

```
[...]

echo "<eiout>"
echo "<eicommands>"
echo "<dialogbox outclass='error' boxtitle='Error!'>"
echo " <content format='html'>"
echo
echo " <ul>"
if [ "$abs" = false ]; then
   echo "<li><span style='color: red'>abs file missing</span></li>"
fi

if [ "$spec" = false ]; then
   echo "<li><span style='color: red'>spec file missing</span></li>"
fi

if [ "$json" = false ]; then
   echo "<li><span style='color: red'>json file missing</span></li>"
fi
echo "</ul>"
echo "</content>"
echo "</dialogbox>"
echo "</eicommands>"
echo "</eiout>"
```

Listato 5.7: Script EI

Con questo codice si crea una finestra popup nel client che indica quali file sono mancanti come in figura.

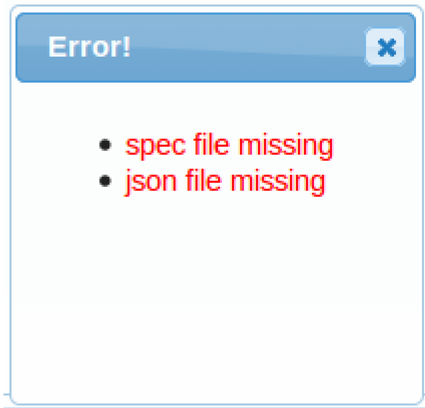


Figura 5.2: Output EI

Infine è necessario applicare una lieve modifica al server per modificare la cartella temporanea dove vengono trasferiti i file in una scelta dall'utente per evitare i problemi con i permessi di scrittura mancanti che non permettono di muovere i file. Questi infatti vanno inseriti nell'apposita cartella per essere tarsferiti come volume a Docker. Nella sezione Files di EIApps.php si può applicare una modifica temporanea per modificare il path della cartella modificando il codice:

```
if ( array_key_exists( '_ei_files', $parameters ) ) {
      $dir = "/etc/EITemp";
      $root_str = $dir;
      EIApps::build_directories($files_str,$dir,$parameters);
      unset( $parameters['_ei_files'] );
    }
```

Listato 5.8: Modifica temporanea a EIApps.php

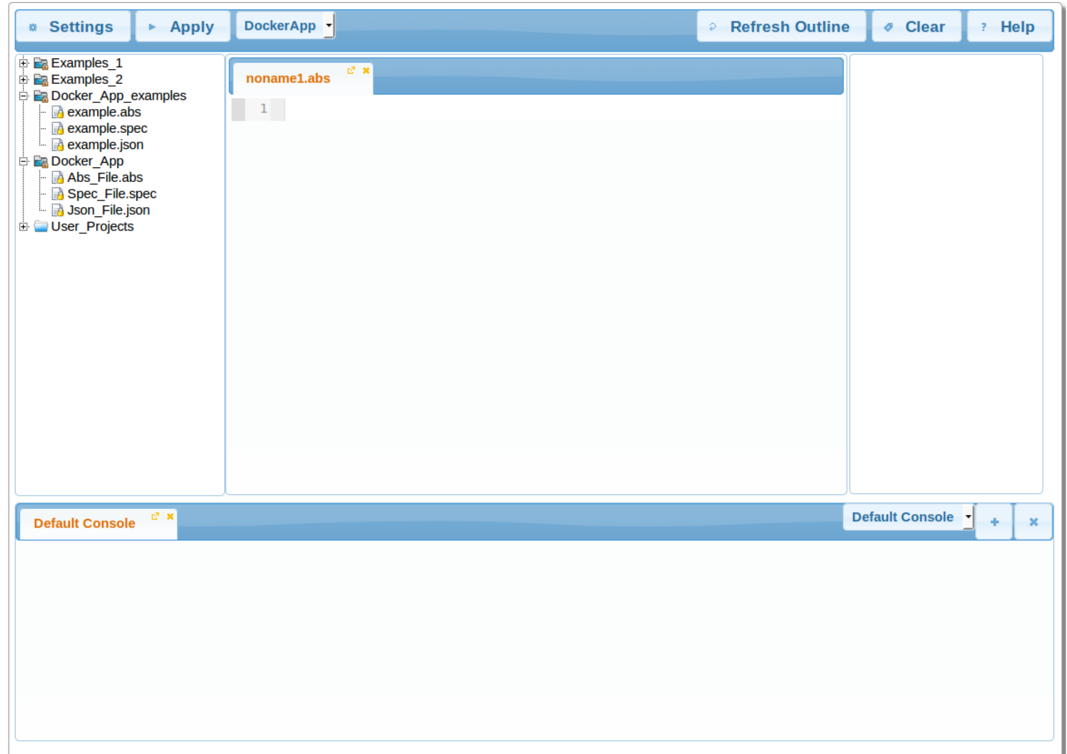
Oltretutto è necessario impostare che l'user www-data (ovvero il server) possa usare il comando "sudo" senza inserire la password, dato che non ha la possibilità di farlo. Aggiungendo nel file sudoers.tmp, il quale può essere modificato con il comando "sudo visudo", la stringa "www-data ALL=(ALL) NOPASSWD: ALL" si otterrà questo risultato. Per aggiungere degli esempi nell'interfaccia si ha un procedimento molto simile a quello per aggiungere applicazioni. Innanzitutto si notifica il server nel file examples.cfg con la stringa:

```
<exset id="Docker_App_examples"
   src="./default/dockerAppExamples.cfg" />
```

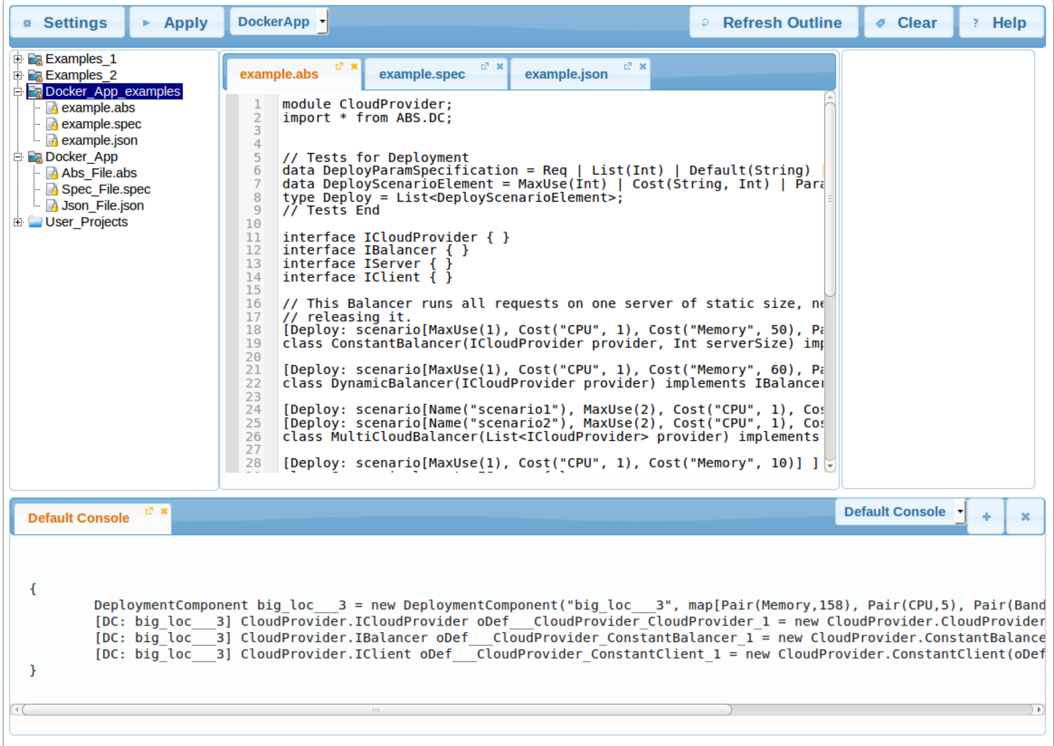
e si crea il file di configurazione come segue:

```
<exset id="Docker_App_examples">
     <folder name="Docker_App_examples">
      <file name="example.abs"
          url="/ei/examples/default/dockerApp/example.abs" />
   <file name="example.spec"
       url="/ei/examples/default/dockerApp/example.spec" />
   <file name="example.json"
       url="/ei/examples/default/dockerApp/example.txt" />
     </folder>
</exset>
```

Completata l'installazione è possibile accedere alla easy interface ed il suo aspetto è mostrato nella figura seguente. Essa mostra l'interfaccia base con gli esempi pre-caricati per poter testare le funzionalità di MODDE. Come si può vedere nella figura b il programma principale di ABS viene scritto come output nella console direttamente.



(a) EI all'avvio



(b) EI dopo aver applicato MODDE

Figura 5.3: Vista della EI

5.4 Sviluppi Futuri

La EasyInterface è progettata per lavorare su un file ABS per volta, in quanto il bottone Apply attiva lo strumento selezionato sul file corrente. Nel futuro per migliorare l'integrazione di MODDE è stata proposta la modifica del client, o la creazione di uno nuovo. Questo per fornire all'utente la possibilità di applicare direttamente sulla cartella desiderata o di gestire quale file mandare in input tramite un menù appropriato. Oltretutto, in quanto come output viene restituito un programma ABS, è prevista la creazione di un ulteriore strumento che renda possibile l'utilizzo diretto dello stesso. Dato che va associato al file ABS fornito come input l'idea principale è permettere l'allocazione direttamente dall'interfaccia. Per esempio il file in input e il suo output potrebbero essere direttamente passati ad uno strumento, diverso per ogni fornitore, ma basato sull'output di MODDE, che lo utilizza per assegnare delle macchine ad un cliente in un sistema Cloud. Per l'uso in modalità offline o nel caso sia necessario effettuare delle modifiche extra sul server è stata creata una macchina virtuale contenente tutte le funzionalità ed è possibile connettersi ad esso tramite una rete interna dal browser del pc in uso. Idealmente per l'uso offline si tende ad utilizzare una macchina virtuale più leggera, possibilmente senza interfaccia grafica in quanto si avvia più velocemente e consuma meno risorse, anche se non garantisce un facile accesso agli utenti generici. Per lo sviluppo di modifiche, invece, un'interfaccia grafica o comunque una macchina virtuale con un sistema operativo completo garantisce un supporto adeguato.

Conclusioni

In questo documento ci siamo concentrati sul proporre un nuovo approccio dove fosse possibile unificare la fase di sviluppo con quella di installazione. Con questo approccio viene permesso all'utente di specificare come deve essere costruito un sistema senza averne delle conoscenze avanzate. Inoltre è stata proposta un'integrazione iniziale di questo strumento in un interfaccia web. Questi strumenti sono ancora in fase di sviluppo, ma questo è un primo passo verso l'automazione dei lunghi processi di studio del sistema che richiedono molto lavoro e una conoscenza dettagliata dell'ambiente di sviluppo. L'obiettivo finale è quello di garantire uno strumento completamente automatico per fornire una soluzione con un click e l'integrazione fornita si basa su questa idea, fornendo una base per i futuri sviluppi. Ovviamente nell'interfaccia vanno inseriti i file di input manualmente anche se potrebbero essere dedotti automaticamente usando strumenti basati su metodi formali. Sfortunatamente questi ultimi non sono ancora abbastanza muturi per essere usati, per questo si usano ancora annotazioni manuali.

Ringraziamenti

Vorrei ringraziare innanzitutto i professori che mi hanno seguito nel corso dello sviluppo della tesi, il prof. Gianluigi Zavattaro e il prof. Mauro Jacopo. Un ringraziamento speciale va a tutti i miei compagni di corso che mi hanno aiutato e sostenuto in questi tre anni e che continueranno nel corso della laurea magistrale: Matteo Aldini, Luca Battistini, Filippo Berlini, Alex Collini, Andrea De Castri, Davide Foschi, Brando Mordenti e Lorenzo Righi. Infine vorrei ringraziare tutte le persone che mi sono state vicine in questi anni che, per motivi di spazio, non posso citare, ma non ne hanno bisogno perché saranno per sempre nella mia memoria.

Bibliografia

[1] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, Alexis Agahi, *Automated synthesis and deployment of cloud applications*, Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, 2014

[2] Tudor A. Lascu, Jacopo Mauro, Gianluigi Zavattaro, *Automatic deployment of component-based applications*, Science of Computer Programming, 2015

[3] Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, Gianluigi Zavattaro, *On the Integration of Automatic Deployment into the ABS Modeling Language* Service Oriented and Cloud Computing, Springer International Publishing, 2015. 49-64

[4] ABS, `http://abs-models.org/`

[5] AEOLUS Project, `http://www.aeolus-project.org/`

[6] Amazon Auto Scaling, `https://aws.amazon.com/autoscaling/`

[7] Amazon EC2, `https://aws.amazon.com/ec2/`

[8] Amazon CloudWatch, `https://aws.amazon.com/cloudwatch/`

[9] Apache, `https://httpd.apache.org/`

[10] Apache Brooklyn, `https://brooklyn.incubator.apache.org/`

[11] Chef, `https://www.chef.io/chef/`

[12] Docker, `https://www.docker.com/`

[13] Easy Interface, `https://github.com/abstools/easyinterface`

[14] MiniZinc, `http://www.minizinc.org/`

[15] OASIS CAMP, https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=camp#technical

[16] OASIS TOSCA, https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

[17] PHP, http://php.net/

Elenco delle figure

1.1 Architettura di ABS 2

2.1 Flusso di esecuzione di MODDE 12

5.1 Differenze fra Docker e macchine virtuali 26
5.2 Output EI 32
5.3 Vista della EI 34