

POLITECNICO DI MILANO
Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica



**KAMI: UN FRAMEWORK
DISTRIBUITO PER MODELLI NON
FUNZIONALI A TEMPO DI
ESECUZIONE**

DeepSE Group
Dipartimento di Elettronica e Informazione

Relatore: Prof. Carlo Ghezzi
Correlatore: Ing. Giordano Tamburrelli

Tesi di Laurea di:
Gianni Bonfanti, matricola 722034

Anno Accademico 2009-2010

*“Scrivi sulla sabbia ciò che dai,
incidi sulla roccia ciò che ricevi.”*

Sommario

I modelli possono aiutare gli ingegneri del software a prendere decisioni di progettazione prima di implementare un sistema. Per costruire tali modelli software, è necessario basarsi su stime numeriche di vari parametri forniti da esperti del settore o estratte da altri sistemi analoghi. Purtroppo, raramente le stime sono corrette. Inoltre, in ambienti dinamici, i valori dei parametri possono variare nel tempo.

In questa tesi verrà discusso un approccio che prende in considerazione questi problemi, monitorando a tempo di esecuzione i modelli e fornendo dati ad uno stimatore, producendo così i parametri aggiornati in tempo reale. L'approccio descritto si focalizza sui modelli per la qualità di servizio, come affidabilità e prestazioni.

Lo scopo di questo lavoro è realizzare KAMI, un framework distribuito che permette di modellizzare sistemi, o parte di essi, in modo da mantenere aggiornati i modelli anche durante l'esecuzione del sistema reale. I componenti sviluppati permettono di monitorare gli input del sistema in esecuzione, di effettuare analisi e verifiche del modello e di catturare violazioni dei requisiti per consentire riconfigurazioni automatiche dell'implementazione. Infine, i contributi della tesi sono stati implementati e validati attraverso un caso di studio e simulazioni sperimentali.

Indice

Sommario	I
Indice	III
Elenco delle figure	V
Listings	VI
1 Introduzione	1
1.1 Breve descrizione del lavoro	1
1.2 Struttura della tesi	2
2 Monitoraggio a tempo di esecuzione di modelli software	3
2.1 Perché usare modelli software	3
2.1.1 Motivazioni	4
2.1.2 Problematiche	5
2.2 Modelli non funzionali	5
3 Stato dell'arte	7
3.1 I progetti di ricerca esistenti	7
3.1.1 Approcci basati sulla misurazione diretta	7
3.1.2 Approcci basati sulla modellizzazione	7
3.2 Peculiarità di KAMI a confronto	9
4 Kami	10
4.1 Architettura	12
4.2 Gestione degli eventi	18
4.2.1 Evento KAMI	19
4.2.2 Dispatcher	19
4.2.3 Hot deployment	21

4.3	Sistemi	22
4.4	Model plugin	26
4.4.1	Realizzazione di un plugin per modelli DTMC	28
4.5	Input plugin	33
4.5.1	Realizzazione di un plugin per input da socket	34
4.6	Reaction plugin	35
4.6.1	Realizzazione di un reaction plugin di notifica	36
4.7	Distribution system manager	36
5	Realizzazioni sperimentali e valutazione	39
5.1	Interfaccia grafica	39
5.2	Caso di studio: TeleAssistance	40
5.3	Scenario di esecuzione	43
6	Conclusioni e sviluppi futuri	48
	Bibliografia	50

Elenco delle figure

4.1	Schema della metodologia di KAMI	11
4.2	Architettura generale di KAMI	13
4.3	Gestori di plugin, sistemi e interfaccia grafica	15
4.4	L'interfaccia Manager implementata dai plugin e la relazione con l'hot deployment	17
4.5	Il pool di KAMI in interazione con le classi Loader, interfacce dei plugin ed eventi	20
4.6	Esempio di sistema DTMC	24
4.7	Scenario di un sistema caricato in KAMI	25
4.8	Scenario di un model plugin caricato in KAMI	27
4.9	Struttura del plugin DTMC	29
4.10	Funzionamento del plugin DTMC	31
4.11	Scenario di un input plugin caricato in KAMI	34
4.12	Il framework in ambiente distribuito	38
5.1	Interfaccia grafica	40
5.2	Processo BPEL TA	41
5.3	Modello DTMC del processo TA	43
5.4	Scenario di configurazione per il processo TA	44
5.5	Requisiti del modello TA calcolati dal DTMC model plugin	47

Listings

4.1	File di configurazione di KAMI (config.xml)	18
4.2	XML Schema Definition per i sistemi KAMI (system.xsd)	22
4.3	Esempio di un sistema KAMI (sistema1.dtmc)	24
5.1	Sistema TA (TeleAssistance.dtmc)	45

Capitolo 1

Introduzione

Un modello software è un'astrazione del sistema rappresentata da uno specifico punto di vista, che cattura gli aspetti importanti del sistema modellizzato semplificando od omettendo il resto. Esso permette di dare una definizione formale dei requisiti, utilizzando valutazioni e simulazioni per generare possibili configurazioni che il sistema può assumere. I modelli possono aiutare gli ingegneri del software a prendere decisioni di progettazione prima di implementare un sistema.

In questo elaborato verrà discusso un approccio che prende in considerazione questi problemi, monitorando a tempo di esecuzione i modelli e fornendo dati ad uno stimatore, producendo così i parametri aggiornati. Il modello aggiornato fornisce una migliore rappresentazione del sistema. Analizzando il modello aggiornato in fase di esecuzione, è possibile rilevare o prevedere se una proprietà richiesta è, o sarà, violata dall'implementazione corrente. Le violazioni dei requisiti possono innescare riconfigurazioni automatiche o azioni correttive volte a garantire gli obiettivi desiderati.

Questa tesi si concentra su modelli software che si occupano di proprietà non funzionali, come affidabilità e prestazioni, e descrive un framework che supporta la metodologia introdotta.

1.1 Breve descrizione del lavoro

Questo lavoro contribuisce a due problemi rilevanti. In primo luogo, esso pone le basi per uno sviluppo iterativo model-driven, che mira a verificare che l'applicazione soddisfi i requisiti non funzionali. Se il

sistema in esecuzione si comporta in modo diverso dalle assunzioni fatte in fase di progettazione, il feedback per il modello mostra perché non soddisfa i requisiti. Questo può portare ad un'ulteriore iterazione di sviluppo o, idealmente, ad azioni di *auto-repairing* che possono generare automaticamente una modifica all'implementazione. In secondo luogo, fornisce una tecnica bayesiana per stimare nuovamente le probabilità, che può essere applicata a diversi modelli formali (come DTMCs, CTMCs o QNs). La metodologia proposta è uno strumento che la sostiene definisce il framework KAMI. KAMI è l'acronimo di *Keep Alive Models with Implementations*.

1.2 Struttura della tesi

La tesi è strutturata nel modo seguente:

Nel Capitolo 2 si introducono le motivazioni che spingono gli ingegneri a utilizzare modelli software, elencando alcune problematiche che questa tesi si propone di risolvere. Si riassume inoltre una panoramica dei modelli non funzionali.

Nel Capitolo 3 si illustra lo stato dell'arte, confrontando la proposta dell'approccio presentato in questo lavoro con i progetti esistenti.

Nel Capitolo 4 si descrive l'architettura del progetto che supporta la metodologia descritta e si espone la struttura dei componenti di KAMI realizzati.

Nel Capitolo 5 si mostra il funzionamento del progetto dal punto di vista sperimentale, riportando uno scenario applicativo concreto in cui la metodologia di lavoro di KAMI risulta efficace.

Nel Capitolo 6 si espongono conclusioni, valutazioni e sviluppi futuri del lavoro effettuato.

Capitolo 2

Monitoraggio a tempo di esecuzione di modelli software

Gli ingegneri del software utilizzano modelli per analizzare il comportamento dei sistemi astraendo dai dettagli. I modelli software sono particolarmente utili nella fase di progettazione per guidare le decisioni architetturali che possono influenzare la qualità complessiva del sistema finale. Utilizzando i modelli, gli ingegneri possono anticipare i difetti che altrimenti verrebbero inseriti nel processo di sviluppo e che condurrebbero successivamente a costose attività di manutenzione. Nella letteratura si possono trovare molti approcci di modellizzazione e molti di loro sono utilizzati nella pratica. Essi si differenziano principalmente per il tipo di proprietà che permettono di modellare e per il livello di precisione dei risultati che è possibile ottenere attraverso essi. In questa tesi, ci concentriamo sui modelli che possono essere utilizzati per analizzare proprietà non funzionali del software-to-be. Inoltre, si occupa di modelli che possono essere utilizzati per la verifica automatica di alcune proprietà.

2.1 Perché usare modelli software

Gli ingegneri utilizzano modelli software in fase di progettazione per identificare le decisioni significative dell'organizzazione di un sistema software, e di definire e stabilire una comprensione condivisa sulle pro-

prietà astratte del sistema. Utilizzare i modelli anche in corso di esecuzione permette di controllare e garantire che il sistema continui a soddisfare i requisiti definiti in fase di progettazione.

Nella pratica comune, gli sviluppatori solitamente non rappresentano le informazioni sui requisiti in modo esplicito in fase di esecuzione. Piuttosto, trasformano e realizzano le proprietà del sistema attraverso implementazioni ad-hoc. Questo non è un problema fintanto che l'insieme dei requisiti del sistema software resta fisso durante l'esecuzione. Tuttavia, in ambienti dove le esigenze degli utenti e le condizioni di funzionamento variano in modo dinamico, un sistema implementato spesso non riesce ad operare in modo adeguato per rispettare un insieme fisso di requisiti. Per questo, nasce la necessità di avere a disposizione sistemi software che possono adattarsi automaticamente alle esigenze degli utenti e ad ambienti operativi mutevoli.

Oggi, l'auto-adattamento è complesso e costoso da implementare e tipicamente viene applicata in settori in cui i sistemi devono fornire il funzionamento continuo o garantire un'alta affidabilità: ad esempio, il software per il controllo delle centrali telefoniche o veicoli spaziali. A seguito della crescente mobilità e la pervasività dell'informatica e della comunicazione, è necessario quindi trovare dei metodi per fare in modo che l'auto-adattamento sia alla portata di sistemi comunemente utilizzati nelle applicazioni di tutti i giorni. Per mantenere l'usabilità, l'utilità e l'affidabilità in tali circostanze, i sistemi devono anche adeguarsi ai cambiamenti degli ambienti. Il progetto di questa tesi si concentra maggiormente a verificare l'adeguatezza dei modelli rispetto a prestazioni e affidabilità.

2.1.1 Motivazioni

Le metodologie di progettazione e sviluppo software sono radicalmente cambiate nel corso degli ultimi dieci anni. I sistemi software sono stati tradizionalmente progettati per operare in un ambiente completamente noto e immutabile. Ogni volta che il software doveva essere modificato, per migliorare la sua qualità o per soddisfare nuove esigenze, era necessario pianificare un nuovo ciclo di vita di manutenzione comprendente progettazione, sviluppo e implementazione di una nuova versione del sistema. Questo approccio porta all'aumento dei costi delle attività di manutenzione e un tempo di time-to-market non adeguato.

Successivamente sono subentrati molti fattori che hanno cambiato l'ingegneria del software negli ultimissimi anni, quali: Internet, la standardizzazione, e i vincoli di time-to-market. Internet ha permesso di sviluppare sistemi software moderni distribuiti geograficamente su larga scala, potendo coinvolgere migliaia di nodi di rete. Inoltre, la standardizzazione ha migliorato l'integrazione del software e i processi di sviluppo, riducendo i costi e aumentando l'affidabilità del software. Infine, cercando di venire incontro ai vincoli di time-to-market, viene prestata maggiore attenzione al riutilizzo del software. Lo scenario appena descritto è noto in letteratura come *Open World Systems* (OWSs) [2].

2.1.2 Problematiche

Il problema chiave dei modelli è la precisione. Intuitivamente, un modello è accurato se le informazioni che il progettista può ricavare forniscono la giusta quantità di dettagli e precisione. Nel caso di requisiti non funzionali, i modelli sono fortemente dipendenti dai parametri che devono essere previsti a priori da parte di esperti di dominio, o sono estratti da altri sistemi analoghi. Purtroppo, raramente le stime sono corrette. Inoltre, i sistemi di grandi dimensioni variano nel tempo. Di conseguenza le ipotesi fatte in fase di progettazione, anche se inizialmente accurate, potranno successivamente cambiare dopo che il sistema è stato distribuito, e anche durante la sua esecuzione.

Per affrontare questi problemi, i modelli devono essere tenuti aggiornati in fase di esecuzione, e devono essere continuamente raffinati per ottenere una precisione sempre migliore, aggiornando i parametri pertinenti. I parametri possono essere aggiornati osservando i dati reali in fase di esecuzione e attraverso una opportuna strategia per raffinare la stima che genera i valori calcolati. Il progetto di questa tesi propone un approccio bayesiano per affrontare questo problema.

2.2 Modelli non funzionali

Prima di introdurre l'approccio proposto da questa tesi (KAMI, vedi Capitolo 4) è necessario riassumere alcuni concetti sui modelli non funzionali e i formalismi che verranno adottati in seguito. Come descritto nell'introduzione, questo lavoro si concentra su affidabilità e prestazioni. Quindi come ingegneri del software ci affidiamo rispettivamente a

modelli di catene di Markov a tempo discreto (*Discrete Time Markov Chains*, DTMCs) e reti di code (*Queueing Networks*, QNs) [4]. È importante notare che sia DTMC che QN sono modelli markoviani: il secondo infatti, può essere ridotto ad un modello di Markov.

I modelli di Markov sono modelli molto generali che possono essere utilizzati per analizzare proprietà relative a prestazioni e affidabilità. Nei modelli markoviani, gli stati rappresentano le possibili configurazioni che il sistema può assumere. Le transizioni tra stati avvengono a tempo discreto o continuo e le probabilità di attivare transizioni vengono descritte da distribuzioni esponenziali. La proprietà di Markov caratterizza questi modelli: vuol dire che, dato lo stato presente, gli stati futuri sono indipendenti dagli stati passati. In altre parole, la descrizione dello stato presente contiene tutte le informazioni necessarie ad influenzare l'evoluzione del processo nel futuro.

I modelli di Markov più utilizzati sono:

- *Catene di Markov a tempo discreto* (DTMC), sono i più semplici modelli markoviani dove le transizioni tra stati avvengono in istanti discreti di tempo. Ad ogni istante (discreto) il sistema può cambiare stato: al tempo i il sistema è in un certo stato, al tempo $i+1$ si può passare ad un nuovo stato;
- *Catene di Markov a tempo continuo* (CTMC), dove il valore associato a ogni transizione uscente da uno stato non è intesa come una probabilità ma come un parametro di una distribuzione esponenziale (*rate di transizione*);
- *Processi di decisione di Markov* (MDP) [17], sono un'estensione dei DTMC che permettono di specificare più comportamenti probabilistici in uscita da uno stato. Questi vengono scelti in modo non-deterministico.

Capitolo 3

Stato dell'arte

3.1 I progetti di ricerca esistenti

Molte tecniche e metodologie supportano la previsione o l'analisi di proprietà non funzionali. In sostanza, due approcci esistenti sono possibili: la *misurazione diretta* e la *modellizzazione*.

3.1.1 Approcci basati sulla misurazione diretta

Il primo approccio è basato sulla misurazione diretta del requisito desiderato di un'implementazione esistente attraverso l'uso di strumenti dedicati (ad esempio, profiler, tracer, ecc.) Ad esempio JMeter [14] esegue profiling di applicazioni Java volte a individuare i colli di bottiglia. Un altro esempio è Load Runner [10], che è stato concepito per eseguire test di carico per le analisi di scalabilità. I dati estratti aiutano ad individuare i componenti critici del sistema che richiedono un perfezionamento per ottenere il comportamento non funzionale desiderato. Se la complessità del sistema aumenta, l'approccio delle misurazioni dirette diventa sempre più difficile; ad esempio, nel caso di sistemi distribuiti.

3.1.2 Approcci basati sulla modellizzazione

La modellizzazione entra in gioco per risolvere i limiti di misure dirette perché può astrarre dalla complessità dei sistemi. Inoltre, un modello può essere costruito prima che un sistema misurabile esista nella realtà. Tuttavia, la pura modellizzazione delle proprietà non funzionali soffre

dei difetti esposti nel Capitolo 2. Di conseguenza la misurazione diretta e la modellizzazione sono due tecniche complementari, piuttosto che alternative.

Per quanto riguarda l'ambito dei web services, molte ricerche già esistenti si concentrano sulla modellizzazione di composizioni di servizi web. Tuttavia la maggior parte di loro si concentrano solo sulle proprietà funzionali. Ad esempio, [11] e [15] descrivono approcci che mirano a verificare e convalidare le composizioni di servizi web utilizzando l'analisi del flusso di lavoro attraverso la verifica dei modelli. Tuttavia il loro approccio non tiene esplicitamente conto di proprietà non funzionali e non sfruttano i dati di a tempo di esecuzione per perfezionare i modelli.

Allo stesso modo, [9] descrive un approccio per verificare composizioni di servizi web a partire dalle descrizioni UML e poi li trasforma in una rappresentazione specifica che consente la convalida rispetto alle proprietà di concorrenza. Un approccio simile è descritto in [8], che mostra come verificare i processi BPEL in caso di limitazioni delle risorse, in relazione alle caratteristiche di *safety* e *liveness*.

Il lavoro in [3] si concentra sul monitoraggio e deriva dati a tempo di esecuzione che vengono analizzati per eseguire la verifica attraverso un linguaggio che utilizza asserzioni. Questo approccio non si basa su un modello esplicito e non supporta la verifica di proprietà non funzionali. Un altro lavoro degli stessi autori [1] definisce un approccio di modellizzazione per composizioni di servizi web e di un linguaggio la cui valutazione si estende anche alla fase di esecuzione. Il linguaggio di asserzioni, chiamato ALBERT, può essere utilizzato per specificare sia le proprietà funzionali, sia le semplici (non probabilistiche) probabilità non funzionali. Le asserzioni ALBERT sono verificate per i workflow BPEL in fase di progettazione attraverso il controllo del modello, e trasformato in affermazioni valutate dinamicamente in fase di esecuzione, una caratteristica che è essenziale per sostenere l'evoluzione in ambienti dinamici.

L'approccio descritto da [18] supporta il monitoraggio on-line dei contratti di servizio (SLA) in un ambiente web-service. Un linguaggio (SLAng) viene introdotto per specificare la qualità del servizio, che comprende gli attributi non funzionali, quali la tempestività, l'affidabilità e il rendimento. Questo approccio è simile all'approccio proposto in questa tesi, perché un modello, basato su automi temporizzati, opera

mentre i messaggi vengono scambiati in fase di esecuzione. La differenza principale è nel principale interesse di questa tesi sulle proprietà probabilistiche e l'adattamento del modello a tempo di esecuzione.

3.2 Peculiarità di KAMI a confronto

La proposta di questo lavoro garantisce sia i vantaggi forniti da approcci basati sulla misurazione e quelle basate su modelli. I modelli sono tenuti aggiornati in fase di esecuzione e, attraverso opportune misure, possono diventare progressivamente più accurati. Solo pochi altri approcci simili sono descritti in letteratura. In particolare, [22] descrive una metodologia per la stima dei parametri del modello attraverso filtri di Kalman. Questo lavoro si basa su un continuo monitoraggio che fornisce dati, in fase di esecuzione, a un filtro di Kalman, volto ad aggiornare il modello di prestazione. Questo approccio però non supporta esplicitamente ambienti dinamici. Inoltre, l'approccio è generale rispetto al modello di prestazione. La proposta di questa tesi invece prevede un sistema statistico specifico che deve essere definito per ogni modello supportato e sviluppato in KAMI tramite plugin.

Il lavoro in [19] descrive una formulazione CTMC di servizi web per prevedere problemi di prestazioni e affidabilità, applicando una tecnica di analisi di sensibilità.

Un recente lavoro [6] presenta un framework per la previsione dell'affidabilità dei componenti il cui obiettivo è di costruire e risolvere un modello stocastico di affidabilità, permettendo a ingegneri del software di visualizzare differenti architetture di progettazione. In particolare, gli autori affrontano la definizione di modelli di affidabilità a livello architettonico e le problematiche relative alla stima dei parametri.

Il problema della corretta stima dei parametri è anche discusso in [12, 20], dove sono individuate le carenze di approcci esistenti e vengono proposte possibili soluzioni.

Per quanto riguarda la tecnica di stima bayesiana a cui il progetto di questa tesi si focalizza, non è conosciuto un approccio esistente che sfrutta questa tecnica statistica per risolvere i problemi presentati in questa tesi. In [21], per esempio, un simile approccio statistico è stato adottato nel predire gruppi di parole per il riconoscimento vocale e di traduzione automatica.

Capitolo 4

Kami

Nel Capitolo 2 è stata introdotta e motivata la necessità per un ingegnere del software di avere a disposizione modelli il più possibile aggiornati e precisi del sistema software in fase di sviluppo. Abbiamo osservato che i modelli sono utili nella progettazione del software, in quanto consentono ai progettisti di integrare le loro precedenti esperienze, la documentazione e le misure in modelli che possono essere analizzati per diagnosticare i problemi ed esplorare le alternative. In particolare, i modelli favoriscono la verifica della conformità tra diverse scelte progettuali e requisiti. Tipicamente vengono forniti modelli differenti per analizzare diversi attributi di qualità e, in particolare, requisiti funzionali e non funzionali. Ad esempio, i modelli per le proprietà non funzionali possono essere utilizzati per prevedere e verificare le prestazioni del software o l'affidabilità. Il lavoro di questa tesi si concentra su questa categoria di modelli, che comprende le reti di code, catene di Markov, Reti Bayesiane, ecc. Come già detto, i progettisti di sistemi verificano se un modello rispetta i requisiti e guidano l'implementazione seguendo la struttura del modello. Se i parametri non corrispondono alla realtà, le prestazioni del software non saranno quelle previste, portando a comportamenti insoddisfacenti o problemi.

L'adattamento a tempo di esecuzione di proprietà non funzionali si propone di risolvere questo problema. Dal momento che i comportamenti previsti del sistema potrebbero differire da quelli attuali, o possono cambiare nel tempo a causa dei cambiamenti dell'ambiente, i modelli per i requisiti non funzionali dovrebbero coesistere con l'implementazione in fase di esecuzione. È quindi possibile alimentare i

modelli con dati a run-time per aggiornare i propri parametri interni. Di conseguenza, i modelli aggiornati forniscono descrizioni sempre più accurate e ci permettono di controllare automaticamente i requisiti desiderati mentre il sistema è in esecuzione. In questa fase, questo approccio può trattare solo l'evoluzione del modello con la stima continua dei suoi parametri numerici. I parametri che possiamo stimare attraverso KAMI rappresentano i valori effettivi delle caratteristiche non funzionali del sistema in fase di progettazione (ad esempio, l'affidabilità di un componente esterno) e del profilo di utilizzo (ad esempio, la distribuzione del tempo di interarrivo dei clienti).

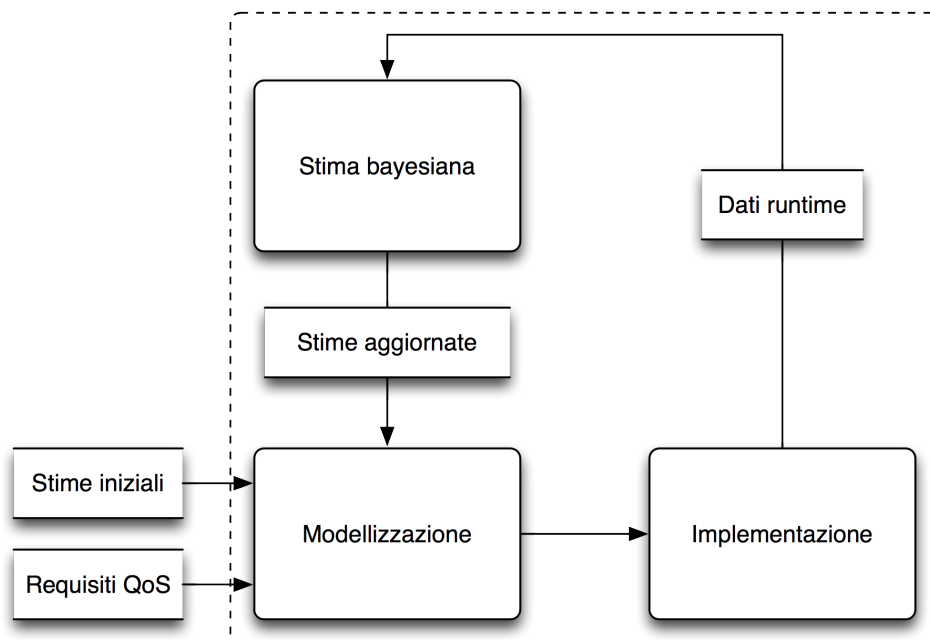


Figura 4.1: Schema della metodologia di KAMI

I vantaggi di questa metodologia sono due. In primo luogo, i modelli aggiornati descrivono meglio i comportamenti sistema reale. In secondo luogo, i modelli aggiornati si evolvono in fase di esecuzione a seguito dei mutamenti nell'ambiente. In entrambi i casi, quando un modello mostra che un determinato requisito è violato, è possibile reagire innescando riconfigurazioni. Inoltre, più i dati vengono raccolti dalle istanze in esecuzione del sistema, più i modelli saranno precisi. In effetti, i parametri del modello convergeranno a valori reali che caratterizzano il sistema modellato. Come mostrato in Figura 4.1, KAMI

stabilisce un circuito di feedback tra modelli e implementazione. In fase di progettazione, i modelli sono sviluppati per verificare i requisiti non funzionali e condurre l'implementazione. In fase di esecuzione, il sistema reale fornisce dati utilizzati come feedback che possono aggiornare il modello, aumentando la sua corrispondenza con la realtà e l'accuratezza. È importante notare che, in KAMI, non è strettamente necessario modellare l'intero sistema, ma solo le sotto-parti che sono considerate critiche.

Un fattore cruciale di KAMI è il meccanismo adottato per trasformare i dati estratti dalle istanze in esecuzione del sistema implementato in stime dei parametri del modello. KAMI svolge questo compito sfruttando tecniche di stima bayesiana.

4.1 Architettura

KAMI è un software basato su plugin. In Figura 4.2 vengono evidenziati i quattro componenti principali del framework, ovvero:

- *Systems*: sono file XML che descrivono i sistemi su cui KAMI opera. Questi file contengono la descrizione del sistema con i parametri numerici che KAMI si occupa di aggiornare e i requisiti a cui l'utente è interessato. Ad esempio, tale file può contenere una descrizione di una rete di code e diversi requisiti (una soglia sul tempo medio di permanenza media o la lunghezza della coda).
- *Model plugins*: forniscono a KAMI la capacità di gestire diversi modelli, attraverso l'interpretazione dei file dei sistemi del tipo supportato. Inoltre, la loro funzione è di verificare che i modelli rispettino i requisiti descritti nel sistema. Se un modello viola un requisito, il model plugin notifica questo evento a KAMI.
- *Input plugins*: forniscono a KAMI la capacità di collegare i modelli con l'ambiente in cui il sistema implementato è in esecuzione. Il sistema in esecuzione alimenta il modello con i dati monitorati. Ad esempio, nel caso di DTMC, fornisce informazioni sull'attivazione di transizioni tra gli stati. Lo scopo del plugin è quello di gestire vari formati di input e protocolli diversi per la gestione a tempo di esecuzione dei dati (ad esempio, socket, RMI, ecc.). Introducono quindi un livello di disaccoppiamento tra KAMI e i

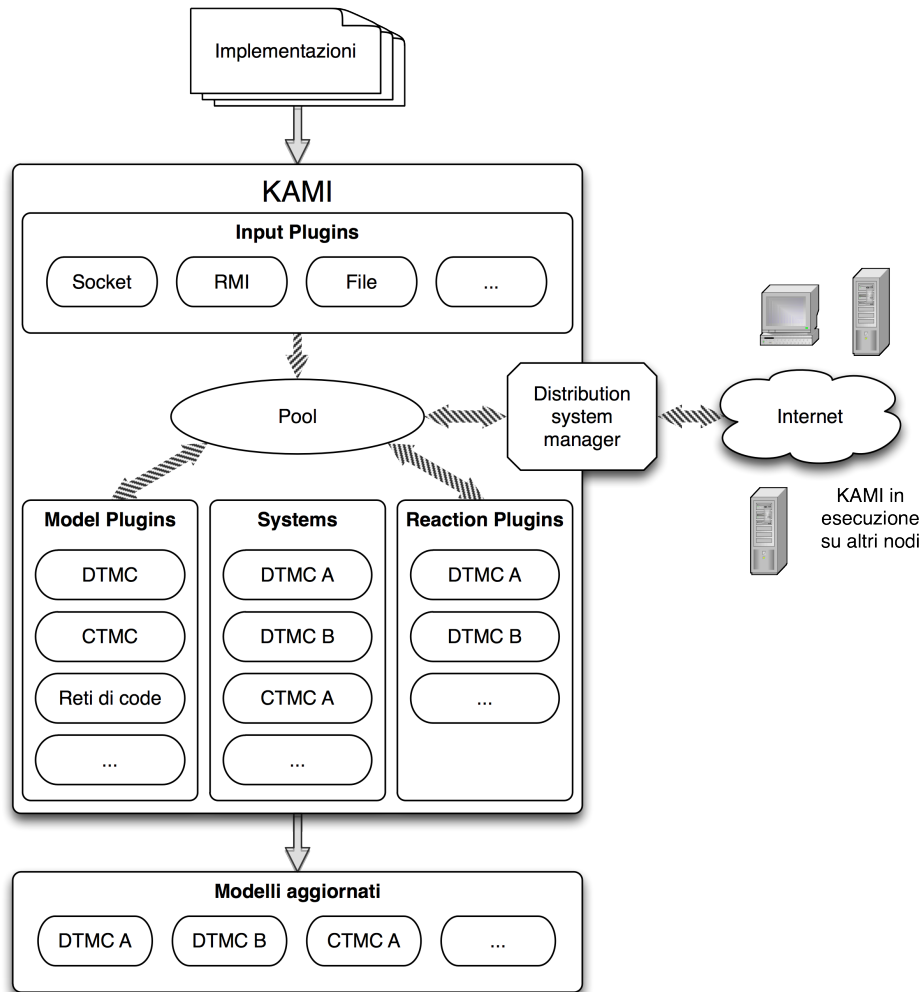


Figura 4.2: Architettura generale di KAMI

sistemi in esecuzione, che permette integrazione e correlazione dei differenti tipi di dati raccolti dal framework sui sistemi.

- *Reaction plugins*: catturano le violazioni che vengono notificate dai model plugin o gli eventi introdotti dagli input plugin e reagiscono attivando delle azioni, ad esempio, per la riconfigurazione del sistema, chiudendo il ciclo di controllo tra il modello e l'implementazione. Tipicamente, i model plugin aggiornano la descrizione del sistema e la verificano rispetto ai requisiti. Il risultato dell'analisi può attivare appropriate reazioni, implementate nei reaction plugin. Possono consistere, ad esempio, nell'invio di un messaggio all'amministratore del sistema, collegare il sistema ad un altro *Web service* o attivare un'altra implementazione.

Il framework è progettato in modo da garantire flessibilità all'utente nella fase di implementazione dei plugin. I plugin sono degli archivi *Jar* che vengono sviluppati *ad hoc* per i tipi di sistemi su cui operare. L'archivio viene poi caricato nel framework con una tecnica di *hot deploy* e il plugin viene verificato e attivato automaticamente attraverso i corrispondenti *manager*. In Figura 4.5 vengono visualizzate le interfacce fornite da KAMI per i tre plugin (Input, Model, Reaction).

Nell'architettura del framework vi sono quattro gestori che governano i componenti principali elencati all'inizio di questa sezione. Ognuno di essi, schematizzati in Figura 4.3, ha la funzione di amministrare il tipo di componente di cui si occupa, gestendone il caricamento, l'eliminazione, la verifica di conformità dei plugin e dei sistemi. Durante l'esecuzione e l'aggiornamento dei modelli, ogni manager ha il ruolo di intermediario tra il dispatcher di KAMI e i plugin stessi, conosce tutte le istanze dei modelli e i sistemi attivi, e viene informato di ogni modifica del file system dall'*hot deploy monitor* associato. Ad esempio, il gestore dei modelli è informato sulle sottoscrizioni di ogni istanza di model plugin che ha lanciato, quindi verrà utilizzato dal dispatcher per notificare correttamente i messaggi.

E' presente inoltre un gestore della *graphical user interface* (GUI), il quale (se attivato), gestisce le operazioni e i comandi impartiti dall'interfaccia grafica. La grafica viene attivata con il seguente argomento della linea di comando:

```
-graphical
```

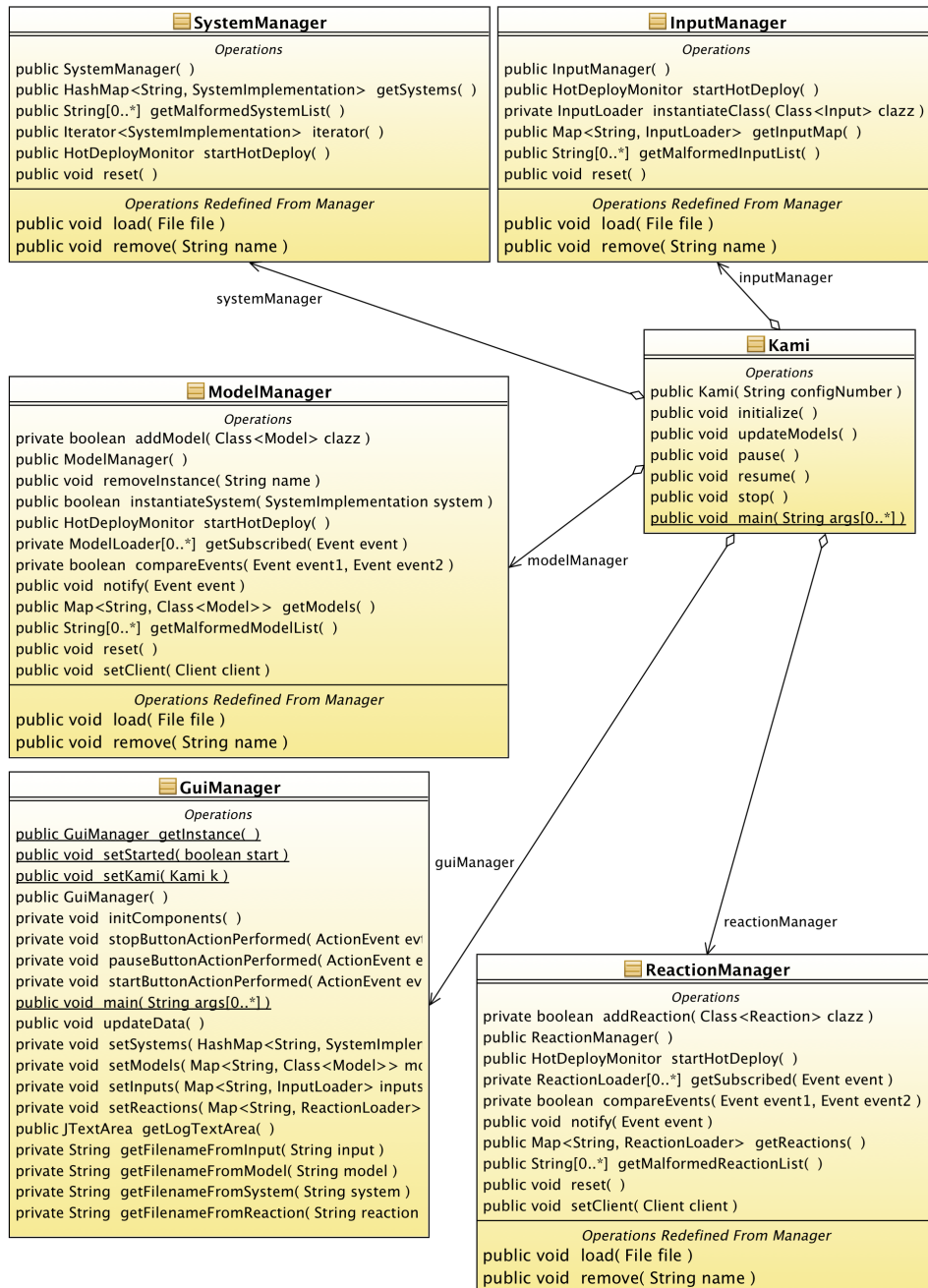


Figura 4.3: Gestori di plugin, sistemi e interfaccia grafica

Come verrà descritto più in dettaglio nella Sezione 4.2.3, ogni gestore è associato ad un *thread* monitor, a cui si affida per la gestione delle modifiche in tempo reale dei file da disco. Questa funzionalità è molto rilevante ad esempio nel caso dei sistemi: KAMI infatti non fornisce un editor per modificare i sistemi, in quanto è possibile farlo con un qualsiasi editor di testo esterno. Il framework individua che il sistema è stato modificato, quindi ferma l'esecuzione dei model plugin associati e ricarica le istanze con i parametri aggiornati del sistema.

A questi quattro componenti base si affianca il *distribution system manager*. Il framework ha la possibilità di collegarsi ad una rete di *broker* KAMI per pubblicare e sottoscrivere messaggi che vengono generati su altri nodi della rete. E' quindi possibile operare in un contesto distribuito, ricevendo informazioni e inoltrando le sottoscrizioni locali a tutti i *peer*.

Il *middleware publish-subscribe* integrato in KAMI è REDS [7] il cui funzionamento verrà descritto nella Sezione 4.7. Non è quindi presente una vera e propria classe manager, ma alcune classi che configurano e attivano *broker* e *client* di REDS, volte a gestire l'integrazione con il middleware.

In Figura 4.4 è raffigurato come l'interfaccia generale Manager viene implementata dai gestori appena descritti.

Altra funzionalità fondamentale del framework è la gestione dei parametri utilizzati dai plugin. All'avvio viene caricata nel sistema una *configurazione* di KAMI, ovvero una serie di parametri che verranno utilizzati da KAMI e dai plugin. Essi sono descritti in un file *config.xml*, che contiene diverse configurazioni. Quando viene lanciato il framework, è possibile specificare quale configurazione caricare nel sistema, attraverso un argomento della linea di comando:

```
-config <configName>
```

I parametri sono descritti come coppie *chiave-valore* e vengono caricate in un oggetto *Properties* del sistema, nella classe *PluginProperties*. I plugin richiamano queste proprietà attraverso il metodo statico:

```
PluginProperties.get(key)
```

dove *key* è la chiave della proprietà che si vuole ottenere.

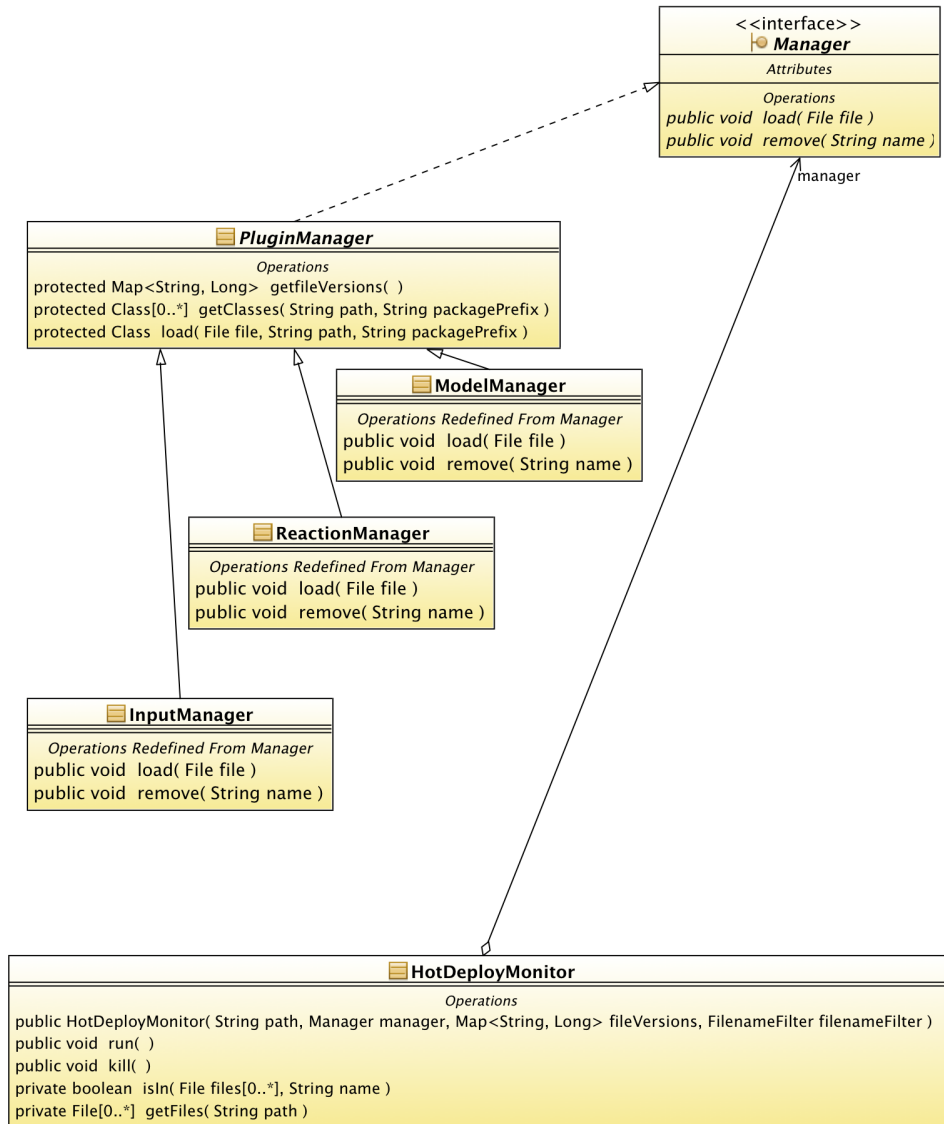


Figura 4.4: L'interfaccia Manager implementata dai plugin e la relazione con l'hot deployment

Un file di configurazione di esempio è il seguente:

```
<?xml version="1.0" encoding="UTF-8" ?>
<config xsi:noNamespaceSchemaLocation="config.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <configuration name="1">
    <key name="ModelCheckingInterval" value="10" />
    <key name="PrismDir" value="prism-3.3" />
    <key name="ServerPort" value="3000" />
    <key name="DistributionSystem" value="1" />
    <key name="RedsLocalPort" value="5000" />
    <key name="RedsRemotePort" value="" />
    <key name="RedsUrl" value="192.168.0.10" />
    <key name="RedsClient" value="false" />
  </configuration>

  <configuration name="2">
    <key name="ModelCheckingInterval" value="7" />
    <key name="PrismDir" value="prism-3.3" />
    <key name="ServerPort" value="3001" />
    <key name="DistributionSystem" value="true" />
    <key name="RedsLocalPort" value="5001" />
    <key name="RedsRemotePort" value="5000" />
    <key name="RedsUrl" value="192.168.0.20" />
    <key name="RedsClient" value="true" />
  </configuration>

  <configuration name="3">
    <key name="ModelCheckingInterval" value="100" />
    <key name="PrismDir" value="prism-3.3" />
    <key name="ServerPort" value="3002" />
    <key name="DistributionSystem" value="true" />
    <key name="RedsLocalPort" value="5002" />
    <key name="RedsRemotePort" value="5000" />
    <key name="RedsUrl" value="192.168.0.30" />
    <key name="RedsClient" value="true" />
  </configuration>
</config>
```

Listing 4.1: File di configurazione di KAMI (config.xml)

4.2 Gestione degli eventi

Prima di procedere con la descrizione del framework è necessario introdurre il concetto di *evento KAMI*, che rappresenta il messaggio fondamentale scambiato tra i vari plugin.

4.2.1 Evento KAMI

Un evento KAMI è una tupla di stringhe di testo. La tupla ha cinque campi:

- *timestamp*: rappresenta data e ora in cui l'evento è stato generato.
- *nome del sistema*: è il nome del sistema a cui l'evento si riferisce e che ha generato l'evento.
- *tipo di evento*: identifica quale tipo di plugin ha generato l'evento. Può quindi assumere i seguenti valori: *input*, *model* e *reaction*.
- *nome dell'evento*: permette di identificare l'evento o il nome del parametro il cui valore è modificato.
- *valore dell'evento*: valore assunto a tempo di esecuzione dal parametro indicato.

Ogni valore della tupla supporta *wildcards*, rappresentato dal carattere asterisco (*) per tutti i campi eccetto il campo del tipo di evento, il quale accetta la stringa ALL. I wildcards vengono utilizzati ad esempio nelle sottoscrizioni dei plugin agli eventi, in quanto hanno la funzione di filtro sui messaggi prelevati dal pool.

Ogni plugin pubblicatore (che può essere input, model o reaction plugin) ha la funzione di incapsulare i dati in questa struttura per notificare i propri eventi a KAMI. Allo stesso modo, ogni plugin sottoscrittore (che può essere solo model o reaction plugin) ricevono eventi da KAMI ed estraggono i dati necessari all'analisi del modello o alla gestione delle riconfigurazioni del sistema implementato.

I plugin vengono eseguiti in thread separati, da una classe chiamata *loader*: queste classi *runnable* hanno il compito di recuperare eventi dal plugin ed inserirli nel pool di KAMI; se sono loader di plugin sottoscrittori, in aggiunta esse prelevano eventi dal pool di KAMI e li memorizzano in una coda locale, in attesa che il plugin li utilizzi. Nello schema di Figura 4.5, sono descritte le interazioni del pool dai vari loader e la struttura dell'oggetto Event.

4.2.2 Dispatcher

KAMI dispone di un meccanismo di comunicazione che permette l'interazione tra plugin, fondato sul paradigma publish-subscribe. I messaggi gestiti dal *dispatcher* del framework sono rappresentati dagli eventi

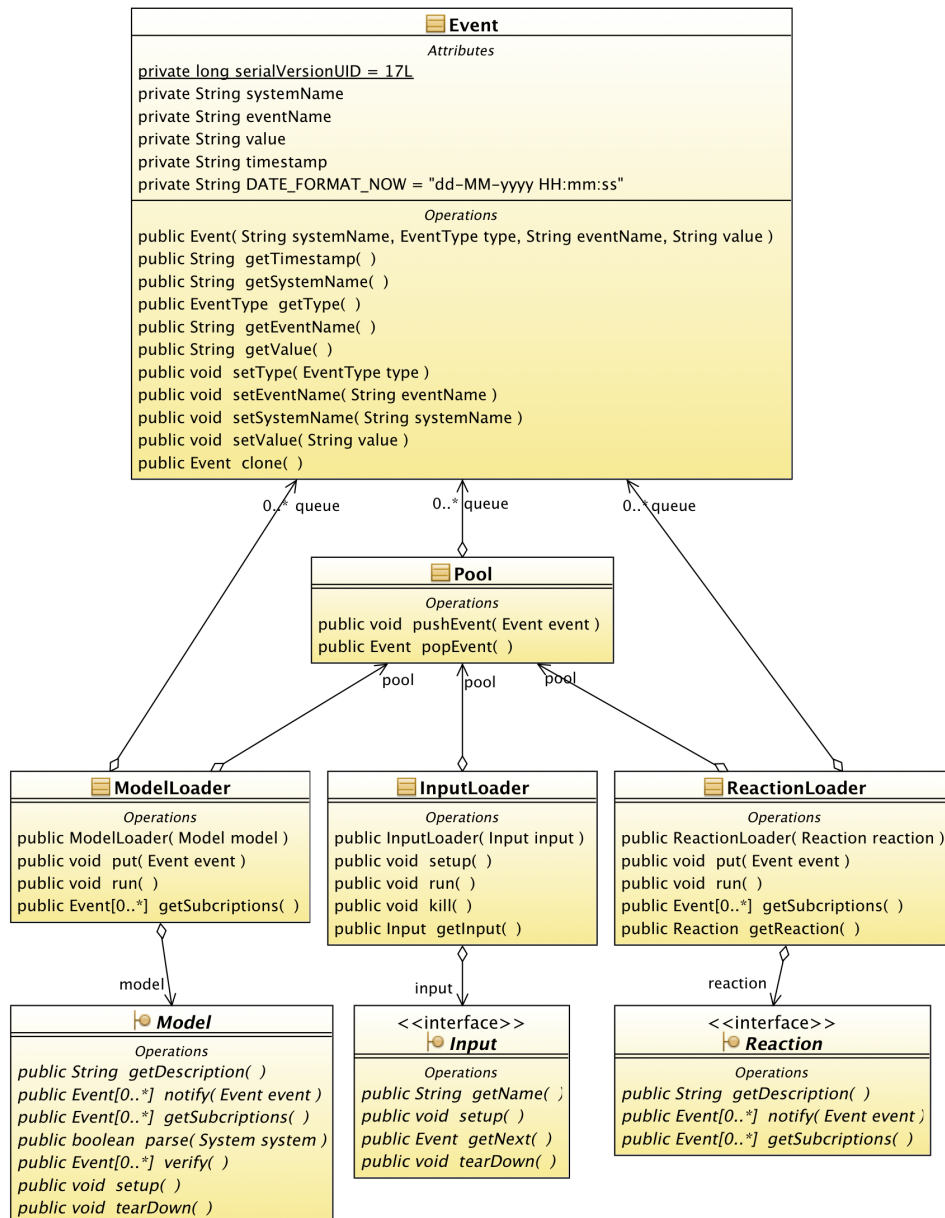


Figura 4.5: Il pool di KAMI in interazione con le classi Loader, interfacce dei plugin ed eventi

KAMI. In questa visione quindi, gli input plugin sono puri pubblicatori; essi pubblicano eventi man mano che ricevono dati dall'istanza del sistema in esecuzione. Model e reaction plugin invece, sono sia pubblicatori che sottoscrittori; i primi sono sottoscritti, ad esempio, a eventi che notificano parametri aggiornati del modello, i secondi invece a messaggi che attivano riconfigurazioni del sistema.

Il dispatcher gestisce quindi il pool interno di KAMI. Ogni evento viene introdotto nel pool dai vari plugin pubblicatori e viene successivamente estratto dal dispatcher e confrontato con la lista delle sottoscrizioni dei model e reaction plugin. KAMI inoltra il messaggio ai plugin interessati tramite i gestori dei plugin, eliminando l'evento dal pool. Se il framework opera in un contesto distribuito, il dispatcher pubblica anche agli altri nodi sottoscritti l'evento prelevato dal pool.

Questo componente è fondamentale in quanto fornisce continuamente dati a tempo di esecuzione che permettono ai model plugin di effettuare stime sul modello e ai reaction plugin di attivare riconfigurazioni, se necessario. Esso può essere messo in pausa o fermato durante l'esecuzione del framework: nel primo caso, il dispatcher continua a prelevare dati dal pool, ma essi vengono scartati e non inoltrati ai plugin sottoscrittori; nel secondo caso il dispatcher viene fermato completamente e tutti i plugin vengono fermati.

4.2.3 Hot deployment

KAMI è progettato per lavorare sempre in un contesto dinamico, dove modelli e sistemi possono cambiare parametri, requisiti e struttura durante l'esecuzione. Per supportare questa metodologia di lavoro, il framework è completamente hot deploy. Esso è sempre attivo, e monitora il file system controllando le quattro cartelle predefinite, relative a input plugin, model plugin, reaction plugin e sistemi.

In questo modo quando un sistema viene modificato, aggiunto o eliminato, KAMI comprende le modifiche in tempo reale e si autoconfigura a caldo. Allo stesso modo vengono gestite le operazioni con i plugin: questo è infatti l'unico modo per aggiungere o rimuovere plugin dal framework. Inoltre, effettua un primo filtro sui file che vengono caricati e rileva le diverse versioni. A seconda del manager a cui è associato, il monitor controlla i file con estensione corretta (Jar o sistemi) e esclude file nascosti o directory.

In Figura 4.4 è rappresentato come l'hot deployment interagisce con le classi manager.

4.3 Sistemi

I sistemi descrivono l'implementazione reale e vengono caricati nel framework effettuando un *pre-parsing* iniziale del file XML. Ogni sistema infatti deve necessariamente essere validato su uno schema XSD predefinito in KAMI per i sistemi e deve quindi contenere le seguenti informazioni:

- nome del sistema
- tipo del sistema
- nome della variabile usata nella descrizione del sistema e valore iniziale
- insieme di requisiti (nome, valore, operatore, soglia)
- descrizione del sistema

La sintassi che regola il nome del file di ogni sistema è *nome.tipo*, dove *nome* e *tipo* si riferiscono al sistema descritto. Il nome del file comprensivo di estensione identifica univocamente il sistema e viene utilizzato nella interfaccia grafica di KAMI per visualizzare e gestire ognuno di essi.

Di seguito è riportato il contenuto del file *system.xsd*, utilizzato da KAMI per validare ogni sistema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="system">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="requirement" minOccurs="1"
          maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="name" type="xsd:ID"
              use="required" />
            <xsd:attribute name="value" type="xsd:string"
              use="required" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

        <xsd:attribute name="operator"
            type="operatorType" use="required" />
        <xsd:attribute name="threshold"
            type="xsd:double" use="required" />
    </xsd:complexType>
</xsd:element>
<xsd:element name="body" type="xsd:anyType"
    minOccurs="0" maxOccurs="1" />
</xsd:sequence>
<xsd:attribute name="name" type="xsd:string"
    use="required" />
<xsd:attribute name="type" type="xsd:string"
    use="required" />
<xsd:attribute name="variable_name"
    type="xsd:string" use="required" />
<xsd:attribute name="init_value" type="xsd:string"
    use="required" />
</xsd:complexType>
</xsd:element>
<xsd:simpleType name="operatorType">
<xsd:restriction base="xsd:string">
    <xsd:enumeration value="&lt;" />
    <xsd:enumeration value="&lt;=" />
    <xsd:enumeration value="&gt;" />
    <xsd:enumeration value="&gt;=" />
    <xsd:enumeration value="=" />
    <xsd:enumeration value="!=" />
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>

```

Listing 4.2: XML Schema Definition per i sistemi KAMI (system.xsd)

KAMI non effettua nessun controllo riguardo alla descrizione del sistema, se non quello di controllarne l'esistenza. Ogni model plugin infatti può avere una differente sintassi e struttura per definire il sistema secondo le necessità dovute alla categoria di modelli a cui appartiene. Esso quindi provvede successivamente ad analizzare la struttura del sistema ed estrarre le informazioni necessarie a costruire il modello.

Ad esempio, nel caso di modelli DTMC, il *body* del sistema contiene la descrizione degli stati e dei valori degli archi che costituiscono il grafo. In Figura 4.6 è rappresentato un semplice esempio di un sistema DTMC con tre stati. Il file XML contenente la descrizione del sistema deve quindi indicare i nomi dei tre stati e i valori delle probabilità delle

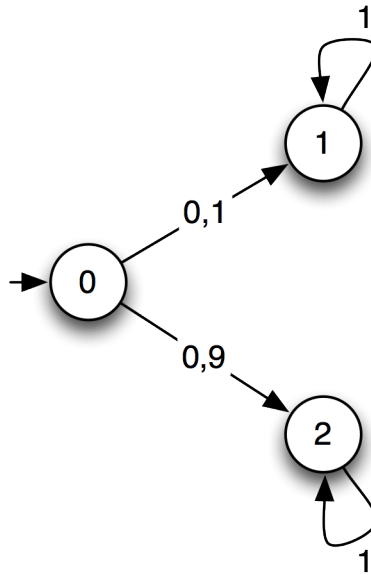


Figura 4.6: Esempio di sistema DTMC

quattro transizioni. Il file del sistema che verrà quindi caricato nel framework sarà quindi il seguente:

```

<?xml version="1.0" encoding="UTF-8" ?>
<system name="sistema1" type="DTMC" variable_name="s"
  init_value="0"
  xsi:noNamespaceSchemaLocation="../config/system.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <requirement name="req1" value="P=? [ F s=1 ]"
    operator=">" threshold="0.3"/>
  <requirement name="req2" value="P=? [ F s=1 ]"
    operator="<" threshold="0.7"/>
  <body>
    <state name="0"/>
    <state name="1"/>
    <state name="2"/>
    <transition source="0" target="1" value="0.1"/>
    <transition source="0" target="2" value="0.9"/>
    <transition source="1" target="1" value="1"/>
    <transition source="2" target="2" value="1"/>
  </body>
</system>

```

Listing 4.3: Esempio di un sistema KAMI (sistema1.dtmc)

Come vedremo in seguito nella Sezione 4.4, il plugin DTMC proposto in questa tesi verifica i modelli a tempo di esecuzione utilizzando un *model checker*. Il plugin utilizza PRISM [16]: di conseguenza, l'attributo "valore" del tag "requirement" contiene la sintassi PRISM per calcolare la probabilità richiesta dal requisito.

Il gestore dei sistemi e il gestore dei modelli hanno una relazione molto forte e i loro ruoli sono spesso dipendenti tra di loro. Ad esempio, in Figura 4.7 sono schematizzate le operazioni di caricamento e rimozione di un sistema nel framework. Il file caricato dall'utente in

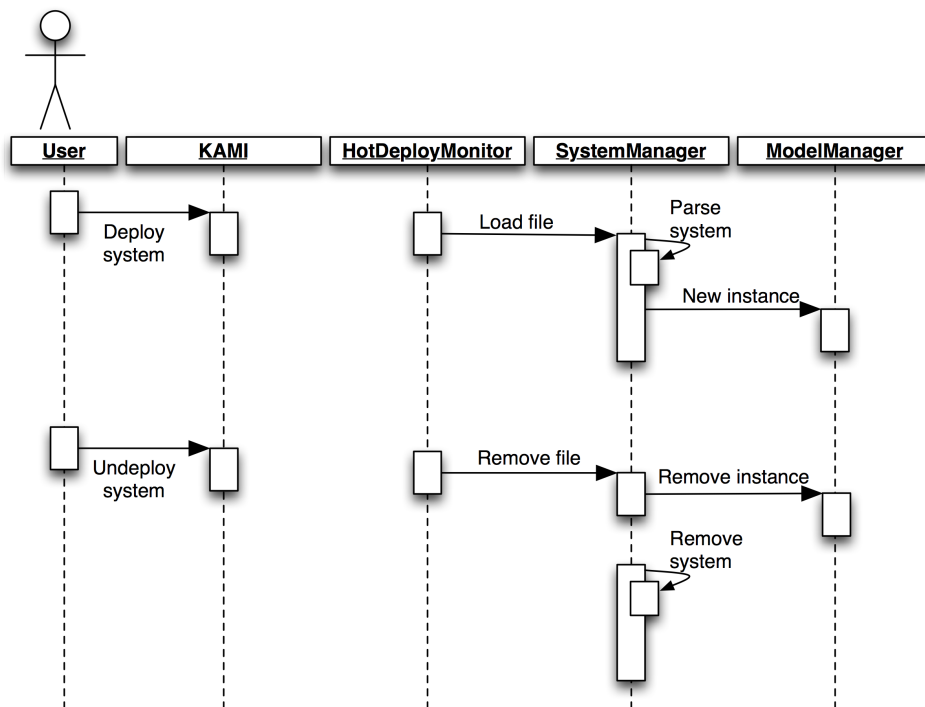


Figura 4.7: Scenario di un sistema caricato in KAMI

KAMI viene notificato dall'hot deploy monitor al manager dei sistemi. Quest'ultimo inizierà una fase di analisi, dove il sistema viene confrontato con lo schema XML definito e viene validato. Successivamente, vengono recuperate le informazioni generiche sul sistema e sui requisiti. Non viene invece analizzata la struttura del sistema, in quanto sarà compito del model plugin.

Se la fase di pre-parsing rileva degli errori, il sistema viene etichettato come *malformato*; altrimenti, il gestore dei sistemi chiede a quello dei

modelli di istanziare il plugin associato al suo tipo (se presente) oppure rimane in attesa che il plugin venga inserito nel framework.

Quando il sistema viene rimosso dall'utente, avviene il procedimento inverso, eliminando il sistema e terminando l'istanza del model plugin in esecuzione.

4.4 Model plugin

Gli ingegneri del software sono incaricati di modellizzare i loro sistemi e sviluppare modelli in KAMI. Ogni model plugin definisce la sintassi che i sistemi sono tenuti a rispettare.

In Figura 4.5 è riportata l'interfaccia Model, a cui ogni plugin dei modelli deve attenersi. Essa infatti prevede l'implementazione dei seguenti metodi:

String getDescription() Restituisce la descrizione testuale del plugin, utilizzata nel log. Viene utilizzato al caricamento del plugin nel framework.

List<Event> notify(Event event) Permette di notificare eventi al plugin. Restituisce una lista di eventi, anche vuota. Viene invocato ogni volta in cui KAMI riceve un evento a cui il plugin è sottoscritto.

List<Event> getSubscriptions() Restituisce la lista di eventi a cui il plugin è sottoscritto. Viene utilizzato quando KAMI riceve un evento.

boolean parse(System system) Verifica la sintassi del sistema ed estrae le informazioni da utilizzare nel modello e per le sottoscrizioni agli eventi. Quando KAMI trova un nuovo sistema e il plugin corrispondente è attivo, il metodo viene invocato.

List<Event> verify() Verifica se il modello viola i requisiti. Restituisce una lista di violazioni, anche vuota. Viene invocato ogni volta che il modello viene aggiornato.

void setup() Contiene le operazioni per inizializzare il plugin, prima di essere avviato. Viene chiamato prima di caricare il plugin.

void tearDown() Contiene le operazioni per terminare il plugin, prima di essere fermato. Viene chiamato prima di eliminare il plugin.

Sono principalmente plugin sottoscrittori, ma hanno la possibilità di pubblicare eventi in risposta ad una notifica, per segnalare ad esempio la violazione di un requisito oppure per attivare analisi più complesse effettuate da altri model plugin.

Lo schema in Figura 4.8 rappresenta, in uno scenario tipico, come un model plugin interagisce con KAMI, assumendo che in esso siano già stati descritti i sistemi. L'utente carica il file Jar del plugin nel fra-

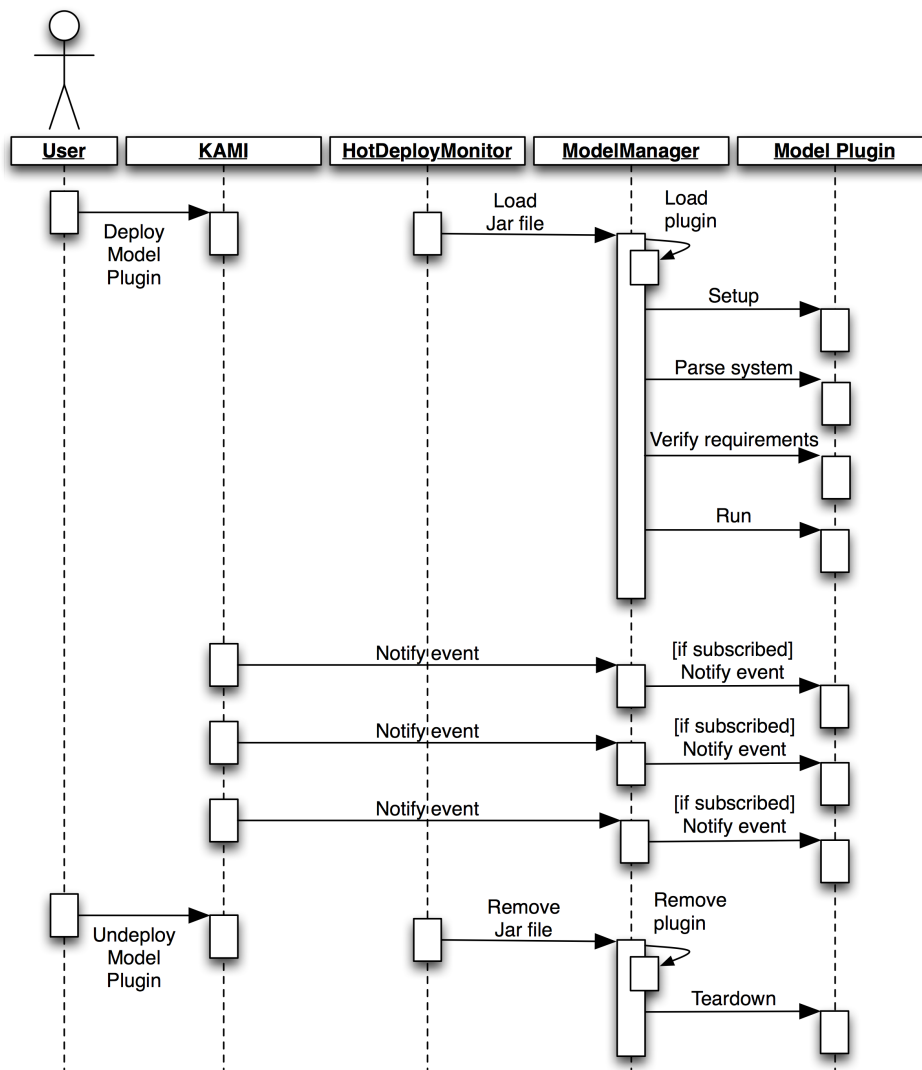


Figura 4.8: Scenario di un model plugin caricato in KAMI

mework e il gestore dei modelli viene notificato dall'hot deploy monitor

dell'aggiunta del plugin. Il model manager carica il plugin e, dopo aver verificato che sia correttamente definito, inizia la fase di configurazione. Essa parte dal *setup* del plugin, prosegue con l'analisi del sistema (o dei sistemi) associati al tipo di plugin corrente e si conclude con la verifica iniziale dei requisiti richiesti. Se tutte queste operazioni vanno a buon fine e non vengono segnalate violazioni, KAMI lancia in esecuzione un'istanza del model plugin per ogni sistema modellizzato.

Successivamente, il plugin viene notificato degli eventi a cui è sottoscritto ed effettua operazioni interne al modello per calcolare i parametri aggiornati. Ad esempio, può ricalcolare probabilità degli stati e frequenze di transizione. In aggiunta, può anche pubblicare eventi nel pool di KAMI.

Quando l'utente decide di fermare il plugin, esso viene rimosso dal file system. L'hot deploy monitor notifica il manager dei modelli dell'eliminazione, il quale provvede a fermare tutte le istanze del plugin in esecuzione. Prima di rimuoverle, effettua le operazioni di chiusura dichiarate nel metodo *tearDown*.

4.4.1 Realizzazione di un plugin per modelli DTMC

Durante questo lavoro è stato sviluppato un plugin per modelli DTMC, il cui diagramma delle classi è riportato in Figura 4.9. Le classi che compongono il plugin sono le seguenti:

DTMC È la classe principale del plugin, che implementa l'interfaccia Model di KAMI. Si occupa principalmente di validare il file e di memorizzare in strutture dati interne le informazioni relative ai requisiti e alla struttura del sistema.

DTMCModel È la classe che esprime il modello vero e proprio. Permette di registrare gli stati e le transizioni del modello iniziale ed espone i metodi per segnalare l'attivazione di una transizione e per richiedere le probabilità delle transizioni aggiornate.

Transition Questo oggetto descrive e contiene le informazioni di una transizione tra stati. Uno degli obiettivi del plugin è di aggiornare queste informazioni con gli eventi ricevuti da KAMI.

PrismParser Questa classe si interfaccia con il model checker probabilistico PRISM. Viene utilizzata dal DTMCModel per effettuare

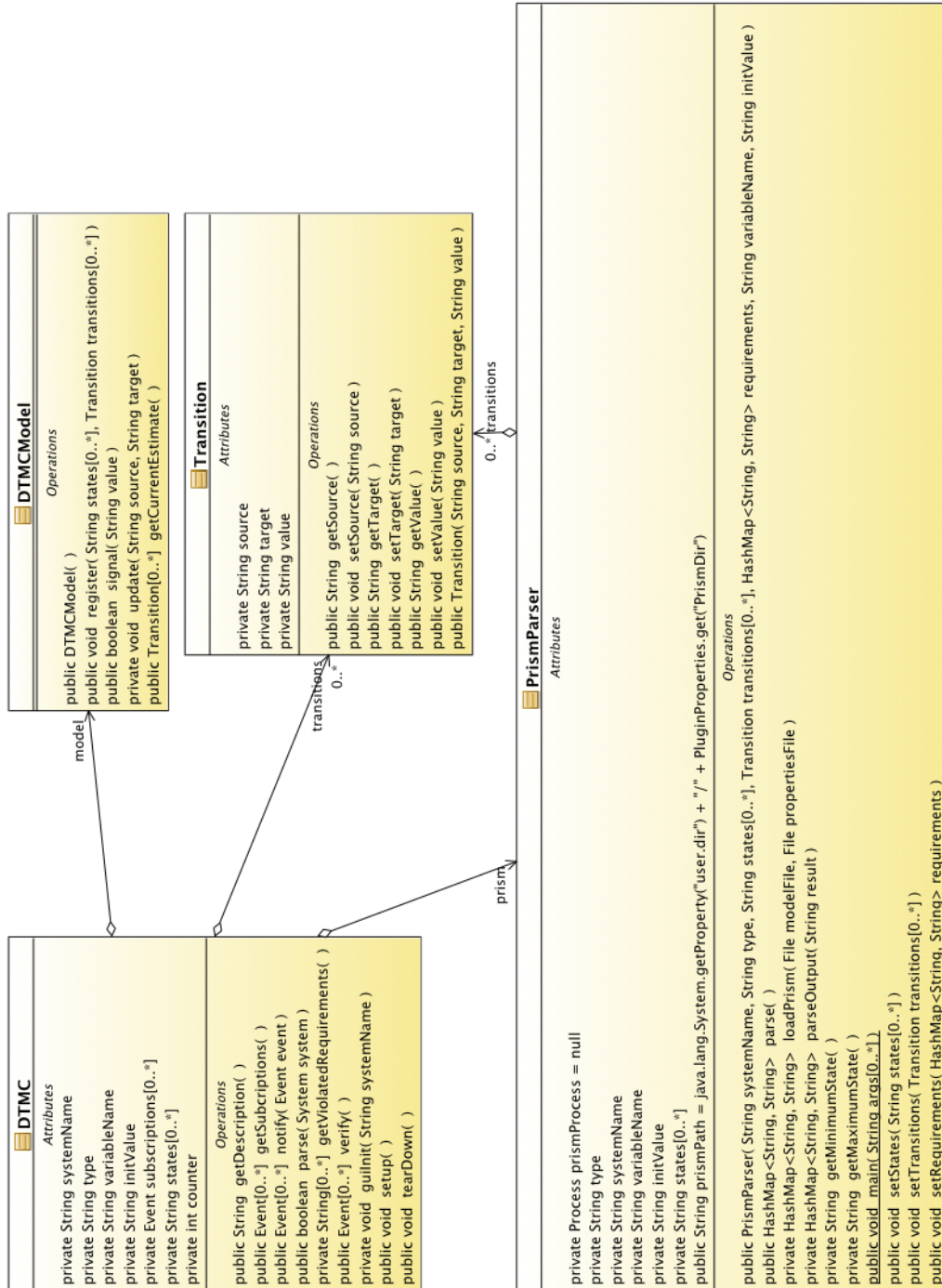


Figura 4.9: Struttura del plugin DTMC

il model checking e calcolare il valore aggiornato delle probabilità delle proprietà richieste dai requisiti.

Il plugin sfrutta alcuni parametri caricati nel framework attraverso il file di configurazione: la directory di installazione di PRISM e la frequenza con cui viene attivato il model checking e la verifica dei requisiti. Inoltre, per ogni istanza del plugin viene visualizzato graficamente una finestra, indipendente dalla grafica di KAMI e gestita completamente dal DTMC model plugin, che contiene le informazioni relative ai requisiti e alle soglie dei parametri non funzionali analizzati. Per ogni requisito, viene mostrato il valore aggiornato calcolato dal model checker, il limite di soglia che deve soddisfare e un box grafico che avvisa attraverso tre colori lo stato del requisito:

- Verde: il requisito è soddisfatto e il valore attuale del parametro è sotto controllo.
- Giallo: indica uno stato di allerta (warning). Il requisito è soddisfatto, ma è molto vicino al limite imposto dal requisito (20% sopra o sotto la soglia).
- Rosso: indica uno stato di errore (error). Il requisito non è soddisfatto.

Per spiegare il funzionamento di questo plugin DTMC, consideriamo lo scenario visualizzato in Figura 4.10: viene caricato in KAMI un sistema DTMC e il plugin viene attivato, cominciando la fase di configurazione. Il sistema viene quindi analizzato dal plugin, validato e vengono estratte le seguenti informazioni dal sistema:

- nome della variabile utilizzata per definire gli stati in PRISM;
- valore assegnato alla variabile in fase di definizione;
- elenco dei nomi degli stati del sistema;
- elenco delle transizioni del sistema, con rispettiva probabilità.

Informazioni come nome, tipo e requisiti sono già conosciute dal plugin e non vengono estratte, in quanto questa operazione viene effettuata precedentemente nella fase di pre-parsing da KAMI. Di conseguenza il plugin crea le *sottoscrizioni* agli eventi da inoltrare al framework. Essendo un modello DTMC, gli eventi a cui il plugin si vuole sottoscrivere sono relativi all'attivazione delle transizioni, per verificare che la

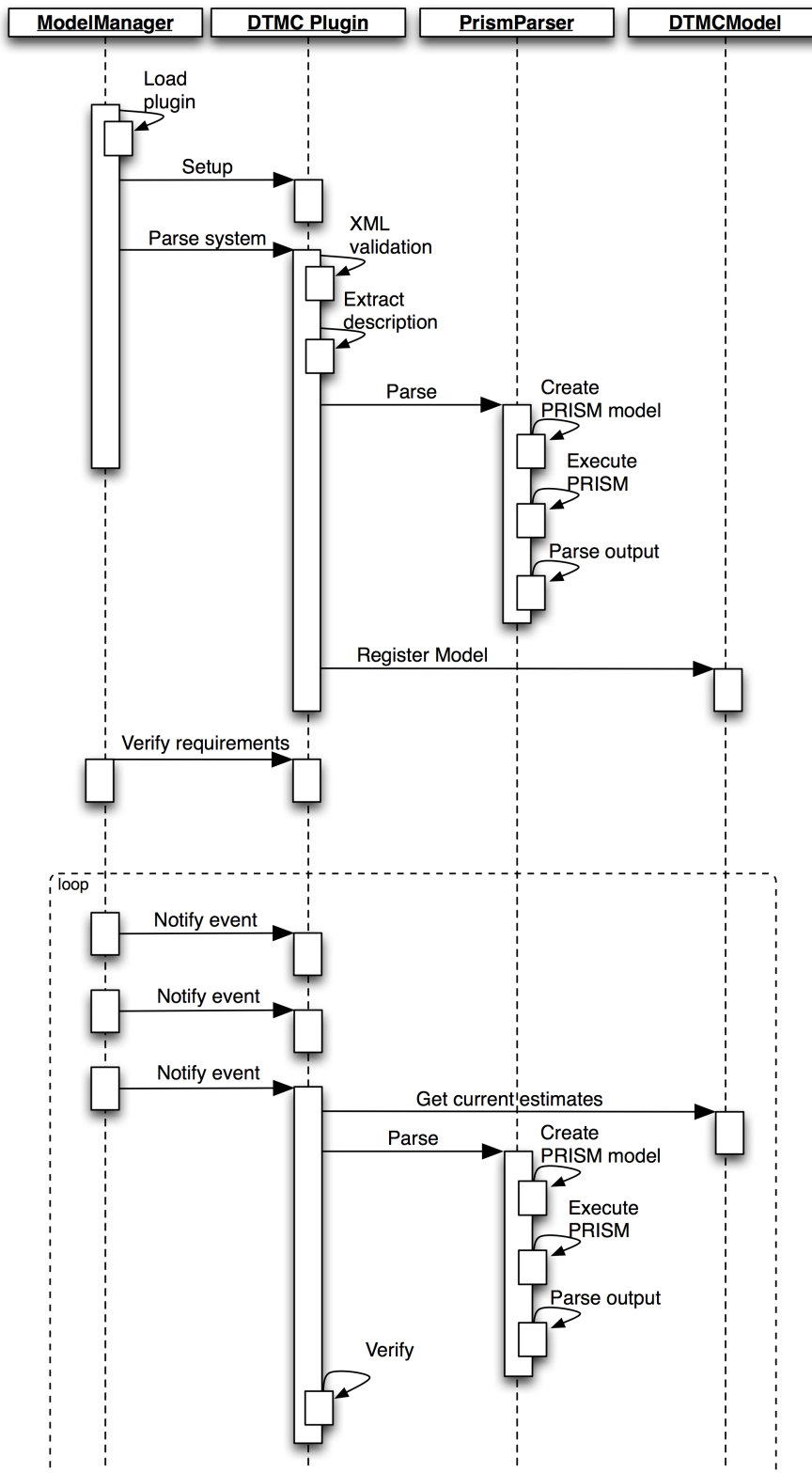


Figura 4.10: Funzionamento del plugin DTMC

probabilità stimata su ogni transizione sia corretta. Le sottoscrizioni quindi saranno formate da una lista di eventi formati da tuple di questo tipo:

```
<systemName, input, transition, source-target>
```

Viene così creata una sottoscrizione per ogni transizione da uno stato *source* a uno stato *target* del sistema specificato, pubblicata da un qualsiasi plugin input.

Successivamente la classe PrismParser crea i file del modello e delle proprietà che il model checker chiede in ingresso. Viene effettuato il model checking e PRISM restituisce le probabilità richieste e specificate nei requisiti del sistema. Non disponendo di API, PRISM viene eseguito in un processo a parte e viene effettuato un parsing dell'output con gestione degli errori, ricavando i valori necessari al modello.

La fase di configurazione prosegue con la registrazione del modello DTMC, memorizzando i dati iniziali e la configurazione del sistema in un oggetto, rappresentato dalla classe DTMCModel che è di fatto il vero modello da aggiornare a tempo di esecuzione. Esso infatti viene aggiornato ad ogni evento ricevuto, e verificato ad intervalli definiti dal file di configurazione caricato. Nell'esempio in Figura 4.10, è stato scelto un intervallo di valore *tre*. Si conclude con la verifica dei requisiti, con i valori di probabilità calcolati dal model checker. Se un requisito non viene rispettato, esso viene notificato a KAMI attraverso un evento formato da una tupla di questo tipo:

```
<systemName, model, violation, requirementName>
```

Terminata la fase di setup, il funzionamento del plugin viene comandato dal gestore dei modelli ogni volta che viene pubblicato un evento a cui il plugin è sottoscritto. Attraverso il metodo notify, il modello viene aggiornato e viene calcolato il valore attuale della probabilità della transizione attraverso metodi di stima bayesiana.

Quando vengono ricevuti un numero di eventi pari all'intervallo definito, in questo caso *tre*, viene invocato nuovamente PRISM come descritto precedentemente, utilizzando i valori delle frequenze delle transizioni del modello stimati dal plugin.

4.5 Input plugin

Gli input plugin hanno il compito di trasformare i dati a tempo di esecuzione in eventi KAMI. Questo meccanismo favorisce l'integrazione di strumenti di monitoraggio esistenti per estrarre i dati dalle istanze di sistemi, perché basta semplicemente sviluppare un input plugin che trasforma i dati del sistema in esecuzione nel formato richiesto dal framework. In Figura 4.5 è riportata l'interfaccia Input, a cui ogni input plugin deve attenersi. Essa infatti prevede l'implementazione dei seguenti metodi:

String getName() Restituisce il nome testuale del plugin, utilizzata nel log. Viene utilizzato al caricamento del plugin nel framework.

void setup() Contiene le operazioni per inizializzare il plugin, prima di essere avviato. Viene chiamato prima di caricare il plugin.

Event getNext() Restituisce il prossimo evento che il plugin vuole pubblicare nel pool di KAMI. E' chiamato da un loop infinito, finché il plugin non viene fermato dall'utente.

void tearDown() Contiene le operazioni per terminare il plugin, prima di essere fermato. Viene chiamato prima di eliminare il plugin.

Lo schema in Figura 4.11 rappresenta, in uno scenario tipico, come un input plugin interagisce con KAMI. L'utente carica il Jar del plugin nel framework e il gestore degli input plugin viene notificato dall'hot deploy monitor dell'aggiunta del plugin. L'input manager carica il plugin e, dopo aver verificato che sia correttamente definito, il plugin viene attivato. Le fasi del ciclo di esecuzione di un input plugin sono essenzialmente tre: (1) fase di setup, (2) fase di esecuzione, (3) fase di chiusura. Le fasi di setup e chiusura sono progettate per permettere al plugin di configurare delle operazioni di preparazione al flusso di dati, che avviene invece nella fase di esecuzione. Ad esempio, possono comprendere operazioni sui file, gestione di socket o configurazione RMI. Gli input plugin sono puri pubblicatori, quindi l'unica funzione della fase centrale è di recuperare dati provenienti dai sistemi in esecuzione e incapsulare le informazioni per pubblicare eventi KAMI.

Quando l'utente decide di fermare il plugin, esso viene rimosso dal file system. L'hot deploy monitor notifica l'input manager dell'eliminazione, il quale provvede a fermare il plugin in esecuzione. Prima

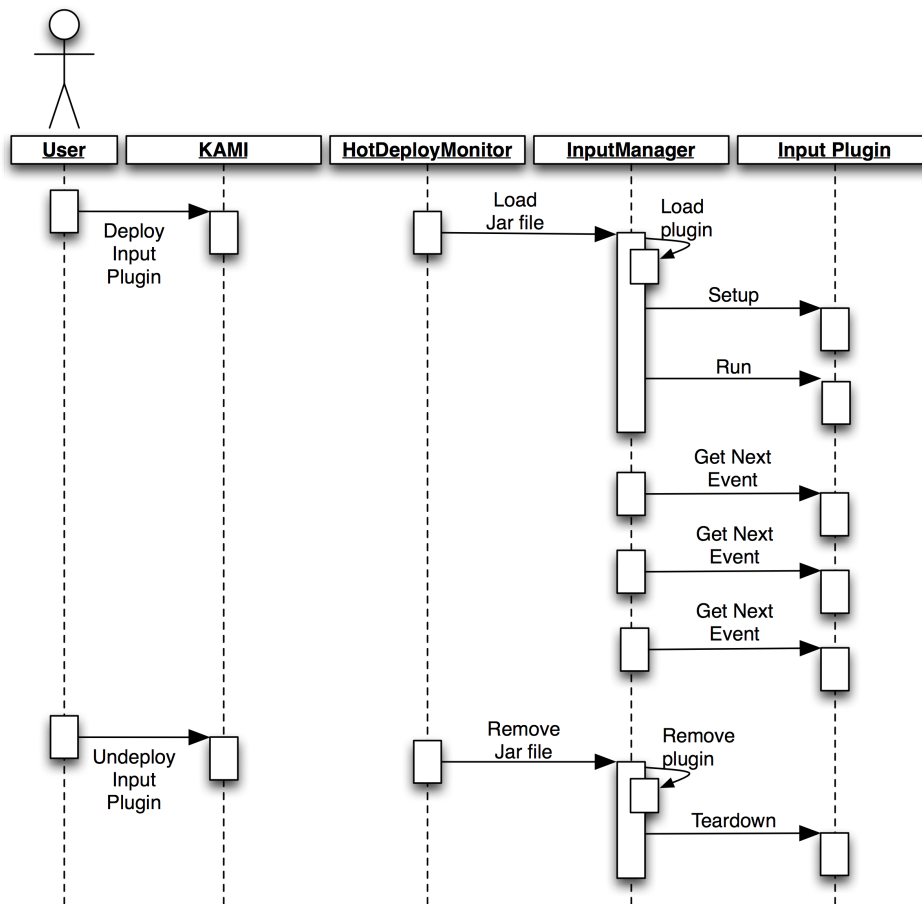


Figura 4.11: Scenario di un input plugin caricato in KAMI

di rimuoverlo, effettua le operazioni di chiusura dichiarate nel metodo `tearDown`.

4.5.1 Realizzazione di un plugin per input da socket

Durante questo lavoro è stato sviluppato un plugin per input provenienti da socket, relativo a sistemi DTMC. Esso è sostanzialmente un socket server che attende connessioni in ingresso su una porta impostata da un parametro di configurazione di KAMI. Il plugin implementa l'interfaccia `Input` nel seguente modo:

String getName() Restituisce il nome testuale del plugin, ovvero *Socket input plugin*.

void setup() Inizializza il server, verificando che la porta assegnata sia libera ed aprendo il socket.

Event getNext() Se il socket è aperto e la connessione con il client è attiva, il plugin estrae i dati e ritorna un evento formato da una tupla di questo tipo:

```
<systemName, input, transition, transitionName>
```

Altrimenti, se il socket non è aperto, il plugin rimane in attesa di connessioni.

void tearDown() Chiude il socket.

4.6 Reaction plugin

I reaction plugin forniscono a KAMI la funzionalità di reagire ai cambiamenti individuati nei sistemi monitorati. La loro principale funzione è di essere plugin sottoscritti agli eventi che innescano riconfigurazioni, ma possono anche pubblicare. Infatti, è possibile configurare diversi plugin in modo da avere catene di reazioni scatenate da reazioni o modelli differenti. In Figura 4.5 è riportata l'interfaccia Reaction, a cui ogni plugin delle reazioni deve attenersi. Essa infatti prevede l'implementazione dei seguenti metodi:

String getDescription() Restituisce la descrizione testuale del plugin, utilizzata nel log. Viene utilizzato al caricamento del plugin nel framework.

List<Event> notify(Event event) Permette di notificare eventi al plugin. Restituisce una lista di eventi, anche vuota. Viene invocato ogni volta in cui KAMI riceve un evento a cui il plugin è sottoscritto.

List<Event> getSubscriptions() Restituisce la lista di eventi a cui il plugin è sottoscritto. Viene utilizzato quando KAMI riceve un evento.

4.6.1 Realizzazione di un reaction plugin di notifica

Durante questo lavoro è stato sviluppato un semplice reaction plugin relativo a sistemi DTMC. Il plugin implementa l'interfaccia Reaction nel seguente modo:

String getDescription() Restituisce la descrizione testuale del plugin, ovvero *Reaction plugin notifier*.

List<Event> notify(Event event) Scrive la ricezione dell'evento nel log di KAMI e manda una mail all'amministratore del sistema in oggetto, segnalandolo della violazione.

List<Event> getSubscriptions() Restituisce la lista delle violazioni a cui il plugin è sottoscritto, riferite ai modelli monitorati.

4.7 Distribution system manager

Finora, il framework è stato descritto in un generico contesto in cui l'istanza viene lanciata in esecuzione in un unico nodo, con i relativi plugin e sistemi, e non comunica con il mondo esterno. Può essere possibile però che l'ambiente richieda di sottoscrivere eventi generati da un'istanza remota, oppure che l'analisi dei modelli e le operazioni di pubblicazione-sottoscrizione dei plugin non avvengano nella stessa macchina.

KAMI integra il middleware publish-subscribe REDS 1.3 (REconfigurable Dispatching System) [7], in aggiunta al motore interno, che permette di creare un dispatcher distribuito attraverso i *broker* della rete e di pubblicare e sottoscrivere eventi attraverso i *client*.

Le funzionalità di *dispatching* sono completamente affidate a REDS. KAMI implementa i due componenti principali:

broker Crea una connessione con la rete di broker KAMI e rimane in attesa per il collegamento di client. Si occupa della distribuzione e l'inoltro dei messaggi ai client sottoscritti. Gestisce i filtri e le sottoscrizioni che transitano nella rete.

client Accede al servizio collegandosi ad un broker, inoltrando le sottoscrizioni ai messaggi a cui è interessato e interagiscono con la rete pubblicando eventi KAMI.

Il distribution system manager utilizza alcuni parametri caricati dal file di configurazione. I parametri sono i seguenti:

DistributionSystem Attiva o disattiva il componente che gestisce REDS. Assume il valore true o false, permettendo quindi di operare in uno scenario locale o distribuito.

RedsLocalPort Definisce la porta su cui il broker si mantiene in attesa di un collegamento da altro broker. È un valore obbligatorio.

RedsRemotePort Definisce la porta del broker remoto verso cui aprire il collegamento. Può anche essere una stringa vuota, indicando che il nodo non vuole connettersi direttamente ad un altro broker.

RedsUrl Definisce l'indirizzo ip del broker remoto verso cui aprire il collegamento. Può anche essere una stringa vuota, indicando che il nodo non vuole connettersi direttamente ad un altro broker.

RedsClient Attiva o disattiva l'interfaccia client del sistema publish-subscribe distribuito. Nel primo caso, KAMI interagisce attivamente con la rete pubblicando e ricevendo messaggi sottoscritti; nel secondo KAMI opera unicamente come broker, ricoprendo il ruolo di nodo dell'infrastruttura della rete.

Le connessioni avvengono attraverso un collegamento TCP e la topologia della rete è genericamente una struttura gerarchica ad albero. In Figura 4.12 è schematizzato il funzionamento del framework distribuito in una rete di tre nodi. Siccome client e broker sono due componenti interni al distribution system manager che vengono eseguiti nella stessa istanza di KAMI, il client è connesso localmente al broker, e non attraverso un collegamento TCP (per ovvie ragioni di prestazioni).

Le sottoscrizioni create all'avvio del framework dai model plugin e reaction plugin vengono quindi inoltrate nella rete dai client attraverso i broker. Successivamente, ogni evento inserito nel pool verrà sia gestito dal dispatcher interno di KAMI che dal dispatcher distribuito di REDS. Allo stesso modo, ogni nuova aggiunta, modifica o eliminazione di plugin o sistemi genera adeguate operazioni di sottoscrizione.

Sfruttando completamente le funzionalità aggiuntive apportate dall'architettura distribuita del framework, è possibile ad esempio operare in uno scenario che effettua l'analisi del modello in un nodo della rete, il monitoraggio del sistema attraverso gli input plugin in un altro nodo

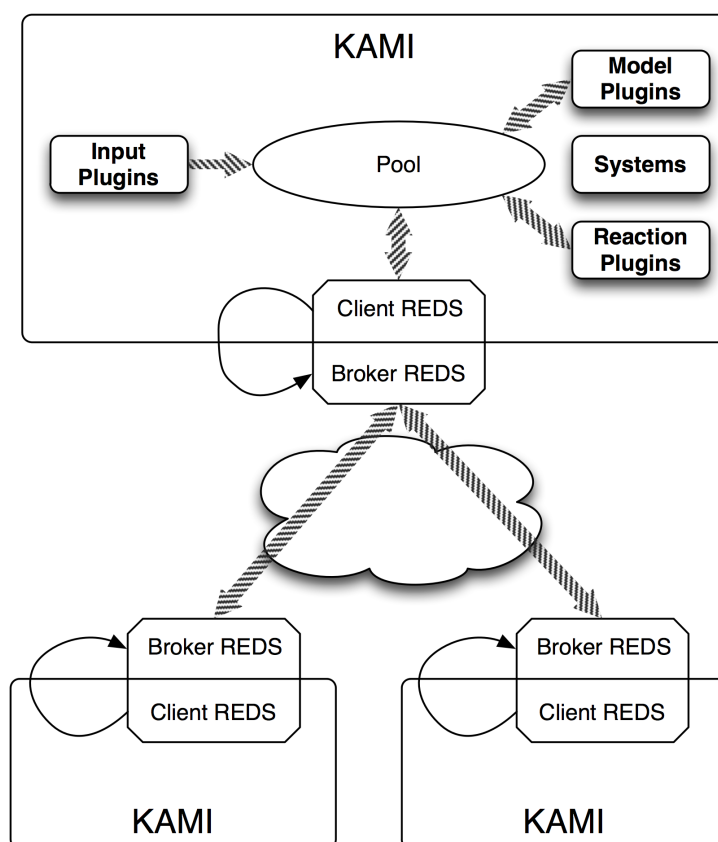


Figura 4.12: Il framework in ambiente distribuito

e avere la gestione delle reazioni in un altro ancora. Oppure il monitoraggio di un sistema può alimentare diversi tipi di analisi di model plugin in esecuzione su istanze di KAMI distribuite nel mondo. O ancora, il model plugin può ricevere informazioni dal sistema monitorato da diversi input plugin dislocati in contesti differenti, per analizzare il tempo di risposta di un servizio web in locazioni diverse.

Uno scenario aperto e distribuito come questo può veramente permettere una concreta modellizzazione del sistema in esecuzione, aumentando la precisione e la concordanza col sistema reale.

Capitolo 5

Realizzazioni sperimentali e valutazione

In questo capitolo vengono mostrate le attività sperimentali svolte e viene illustrato il funzionamento del framework, in uno specifico sistema reale distribuito. Viene inoltre introdotto il caso di studio su cui si basa lo scenario di esecuzione.

5.1 Interfaccia grafica

L'interfaccia grafica di KAMI permette all'utente di svolgere operazioni e visualizzare informazioni riguardanti a plugin e sistemi. In Figura 5.1 è riportata l'interfaccia durante l'avvio del framework in una configurazione vuota. Nella metà in alto della finestra si trovano gli elenchi dei plugin e dei sistemi caricati nel framework. A destra, viene presentata la *libreria* di KAMI, ovvero la raccolta di plugin e sistemi pronti per essere avviati. Questa parte superiore dell'interfaccia gioca ruolo fondamentale nell'interazione dell'utente con il sistema, in quanto attraverso i menu contestuali del mouse è possibile effettuare *deploy* e *undeploy* dei plugin e sistemi.

Nella metà in basso invece, si trova il *log* contenente ogni operazione effettuata sia dal framework che dai plugin. KAMI infatti espone le API per permettere ai sviluppatori di plugin di scrivere messaggi nel log di sistema.

Infine, nella barra in basso, sono presenti gli unici pulsanti per gestire l'esecuzione del framework: due pulsanti di avvio/interruzione e un pul-

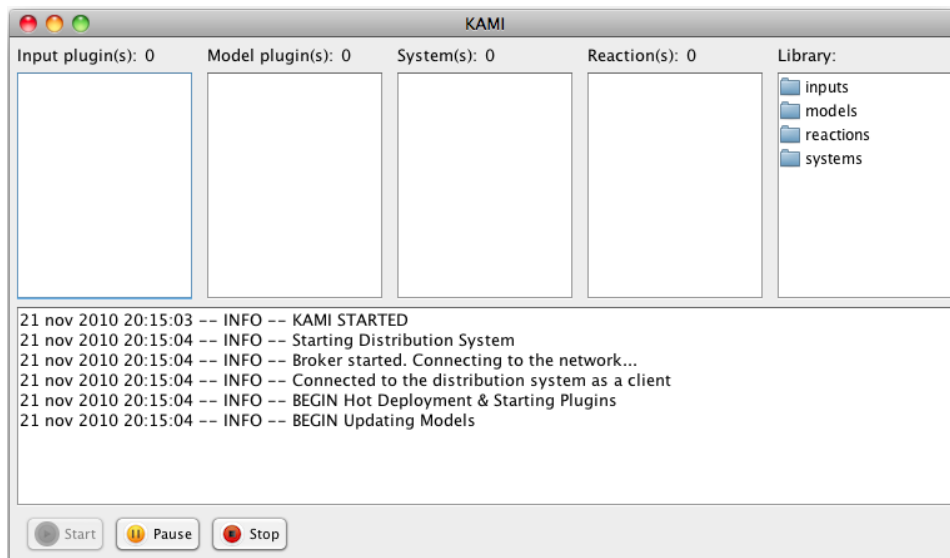


Figura 5.1: Interfaccia grafica

sante di pausa, che consente di sospendere temporaneamente KAMI e riprendere l'esecuzione successivamente. Mettere in pausa il framework (in un istante $t=i$) significa fermare l'aggiornamento dei modelli: i plugin rimangono in esecuzione, mentre il motore interno di publish-subscribe viene fermato, con conseguente eliminazione degli eventi che i plugin richiedono eventualmente di pubblicare. Ripristinando l'esecuzione (in un istante $t=i+p$, con p corrispondente alla lunghezza della pausa), KAMI riparte con l'aggiornamento dei modelli, gestendo gli eventi dall'istante successivo a $t=i+p$.

5.2 Caso di studio: TeleAssistance

In questa sezione viene descritto un caso di studio (una composizione di Web-service) utilizzato in questo capitolo per illustrare un esempio concreto di utilizzo del framework. Nel caso di *Service Oriented Systems* (SOAs) l'esigenza di mantenere modelli aggiornati a tempo di esecuzione è molto rilevante. Una composizione di Web-service è un'orchestrazione di servizi volti a costruire un nuovo Web-service che sfrutta un insieme di servizi già esistenti. L'orchestrazione è svolta attraverso il linguaggio BPEL [5]. Le istanze BPEL coordinano servizi che sono in genere gestiti da organizzazioni esterne, oltre al propieta-

rio della composizione di servizi. Questa gestione distribuita implica che le proprietà funzionali e non funzionali del servizio finale si basano sul comportamento di partner di terze parti che influenzano i risultati ottenuti. In fase di progettazione, un modello può essere utilizzato per garantire che la qualità di servizio (QoS) di un servizio composto soddisfi i requisiti, sulla base della QoS ipotizzata di ciascun servizio esterno. Tuttavia, la verifica durante la fase di progettazione non è sufficiente. Il QoS dichiarato dalla composizione di servizi può rivelarsi non soddisfatto in pratica. Inoltre, a causa della natura decentrata di servizi e proprietà distribuita, i servizi esterni possono subire variazioni indipendenti e impreviste, che possono portare a violare le esigenze globali di QoS.

L'esempio che viene utilizzato in questo lavoro è basato su un caso di studio, illustrato in [1], che si occupa di un sistema distribuito per l'assistenza medica. L'applicazione, chiamata TeleAssistance (TA), consiste in un processo BPEL per l'assistenza remota dei pazienti. La Figura 5.2 illustra l'applicazione, in cui un server esegue il servizio composto TA. La descrizione è fornita graficamente.

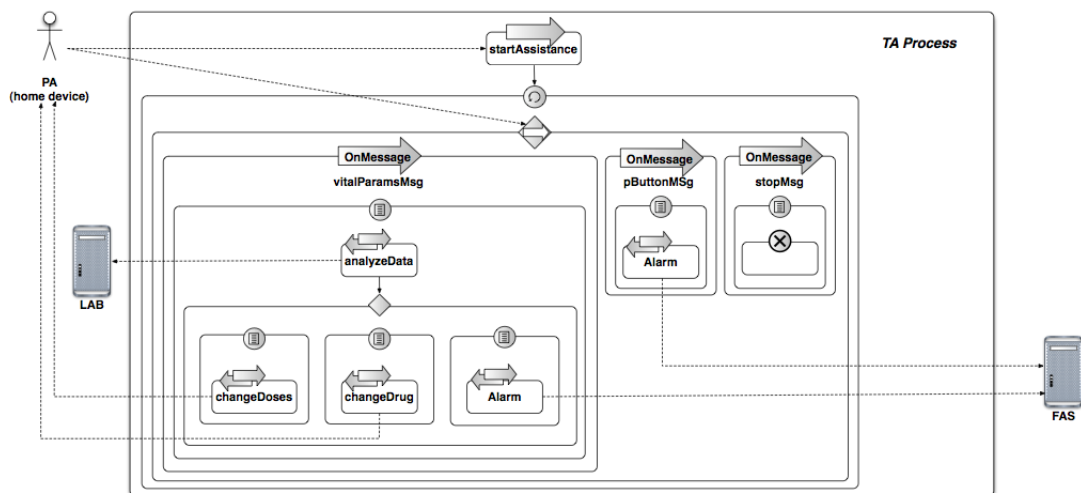


Figura 5.2: Processo BPEL TA

Il processo inizia non appena un paziente (PA) abilita il dispositivo fornito da TA, che invia un messaggio all'attività "receive" del processo *startAssistance*. Successivamente si entra in un ciclo infinito dove ogni iterazione è formata da un'attività "pick" che sospende l'esecuzione e

attende uno dei seguenti tre messaggi:

1. *vitalParamsMsg*: contiene i parametri vitali del paziente che vengono inoltrati dal processo BPEL al servizio medico di laboratorio (LAB), richiamando l'operazione *analyzeData*. Il LAB ha il compito di analizzare i dati e risponde inviando un risultato memorizzato in una variabile chiamata *analysisResult*. Un campo della variabile contiene il valore di risposta che può essere: *changeDrug*, *changeDoses* o *sendAlarm*. L'ultimo di questi messaggi innesca l'intervento di una squadra di primo soccorso (FAS), composta da medici, infermieri e paramedici, il cui compito è quello di visitare il paziente a casa in caso di emergenza. Per allertare la squadra, il processo TA invoca l'operazione di allarme del FAS.
2. *pButtonMsg*: genera e invia un allarme al FAS, causato dalla pressione del pulsante di panico del paziente.
3. *stopMsg*: indica che il paziente decide di annullare il servizio di teleassistenza.

Infine, si presuppone che il progettista del sistema sia interessato a verificare i seguenti requisiti di affidabilità:

- **R1**: la probabilità P1 che si verifichino guasti è minore di 0,3
- **R2**: se si verifica un'attività *changeDrug* o *changeDoses*, la probabilità P2 che il prossimo messaggio ricevuto dal TA genera un allarme che fallisca (il FAS non viene notificato) è inferiore a 0,015
- **R3**: assumendo che gli allarmi generati da *pButtonMsg* abbiano una bassa priorità, mentre gli allarmi generati da *analyzeData* abbiano alta priorità, la probabilità P3 che un allarme ad alta priorità fallisca (non è notificato al FAS) è inferiore a 0,012

Volendo monitorare parametri di affidabilità del sistema TA, il modello DTMC è quello più appropriato per verificare i requisiti richiesti. L'attività di modellizzazione consiste nell'identificazione degli stati rilevanti del sistema assegnando adeguate probabilità alle transizioni tra stati e probabilità di fallimento durante le invocazioni dei servizi. In Figura 5.3 è rappresentato il modello DTMC del processo TA. È importante notare come i parametri del modello possano essere forniti da esperti di dominio, sistemi analoghi già esistenti o precedenti versioni del sistema in fase di progettazione. In ogni caso, questi valori

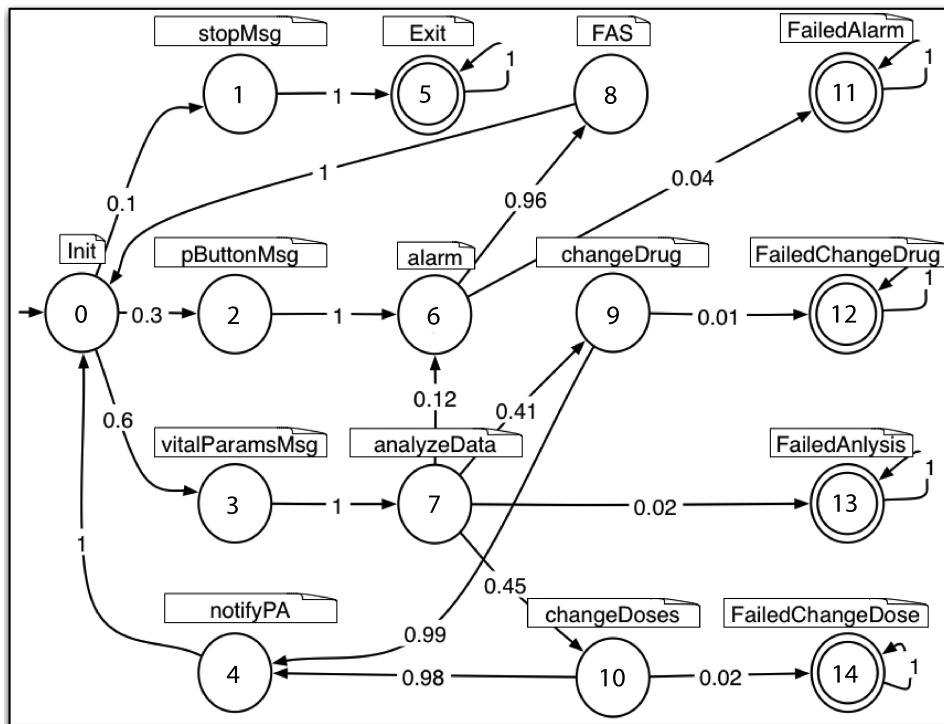


Figura 5.3: Modello DTMC del processo TA

rappresentano solo una stima dei parametri e l'analisi a tempo di esecuzione effettuata da KAMI ha l'obiettivo di perfezionarli attraverso un continuo monitoraggio del modello e verifica dei requisiti.

5.3 Scenario di esecuzione

KAMI viene eseguito in tre istanze separate su tre nodi come schematizzato in Figura 5.4, in cui ricopre ruoli differenti:

monitor Si interfaccia con il server di monitoraggio del sistema TA attraverso il socket input plugin descritto nella Sezione 4.5.1, ricevendo eventi durante l'esecuzione del servizio web. Viene monitorato ogni cambiamento di stato del sistema in esecuzione, in modo da creare eventi che indicano l'attivazione di una transizione nel modello DTMC.

modeler Racchiude le operazioni di analisi del sistema e di aggiornamento del modello, attraverso il DTMC model plugin descritto

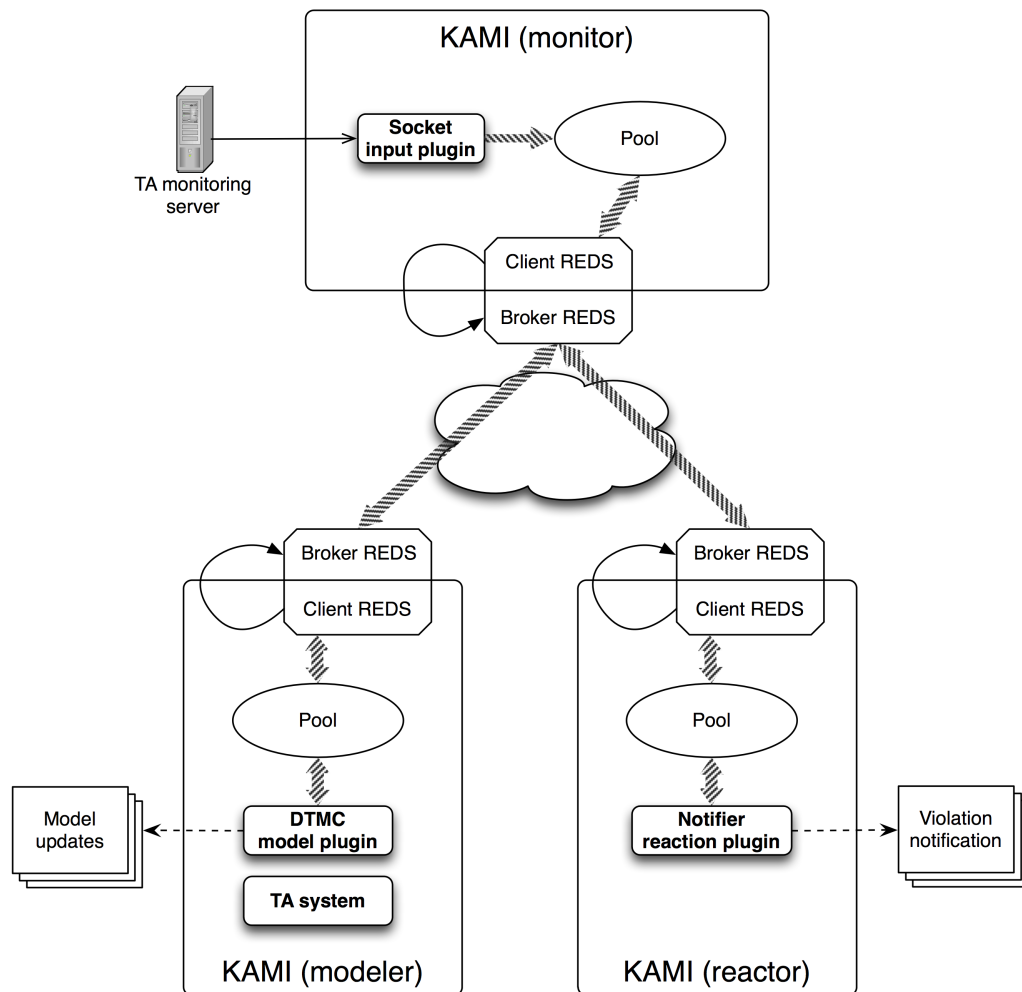


Figura 5.4: Scenario di configurazione per il processo TA

nella Sezione 4.4.1. Analizza i requisiti descritti nel file *TeleAssistance.dtmc* e costruisce il modello, lo mantiene aggiornato attraverso gli eventi a cui è sottoscritto e che vengono pubblicati dall'istanza di KAMI di monitoraggio, e invoca periodicamente il model checker probabilistico PRISM per calcolare le probabilità dei requisiti richiesti. Se necessario, pubblica eventi relativi alla violazione dei requisiti descritti nel sistema.

reactor Si sottoscrive, attraverso il notifier reaction plugin descritto nella Sezione 4.6.1, ad eventi che notificano le violazioni dei requisiti, provenienti dal model plugin. Visualizza le proprietà

non verificate e notifica l'utente con informazioni inerenti alla riconfigurazione del sistema.

Dopo aver avviato i plugin nelle rispettive istanze di KAMI, è necessario descrivere il sistema da monitorare secondo la sintassi stabilita dal model plugin. Di seguito è riportato il file del sistema TeleAssistance: esso implementa il modello DTMC rappresentato in Figura 5.3.

```
<?xml version="1.0" encoding="UTF-8" ?>
<system name="TA" type="DTMC" variable_name="s"
  init_value="0"
  xsi:noNamespaceSchemaLocation=" ../ config/system.xsd"
  xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance">

  <requirement name="R1" value="P=? [ true U s>10 ]"
    operator="<" threshold="0.3" />
  <requirement name="R2" value="P=? [ s!=1 & s!=3 & s!=8 U
    s=11{s=4} ]" operator="<" threshold="0.015" />
  <requirement name="R3" value="P=? [ s!=8 & s!=2 U
    s=11{s=7} ]" operator="<" threshold="0.012" />

  <body>
    <state name="0" />
    <state name="1" />
    <state name="2" />
    <state name="3" />
    <state name="4" />
    <state name="5" />
    <state name="6" />
    <state name="7" />
    <state name="8" />
    <state name="9" />
    <state name="10" />
    <state name="11" />
    <state name="12" />
    <state name="13" />
    <state name="14" />
    <transition source="0" target="1" value="0.1" />
    <transition source="0" target="2" value="0.3" />
    <transition source="0" target="3" value="0.6" />
    <transition source="1" target="5" value="1" />
    <transition source="2" target="6" value="1" />
    <transition source="3" target="7" value="1" />
    <transition source="4" target="0" value="1" />
    <transition source="5" target="5" value="1" />
```

```

<transition source="6" target="8" value="0.96" />
<transition source="6" target="11" value="0.04" />
<transition source="7" target="6" value="0.12" />
<transition source="7" target="9" value="0.41" />
<transition source="7" target="13" value="0.02" />
<transition source="7" target="10" value="0.45" />
<transition source="8" target="0" value="1" />
<transition source="9" target="12" value="0.01" />
<transition source="9" target="4" value="0.99" />
<transition source="10" target="14" value="0.02" />
<transition source="10" target="4" value="0.98" />
<transition source="11" target="11" value="1" />
<transition source="12" target="12" value="1" />
<transition source="13" target="13" value="1" />
<transition source="14" target="14" value="1" />
</body>
</system>

```

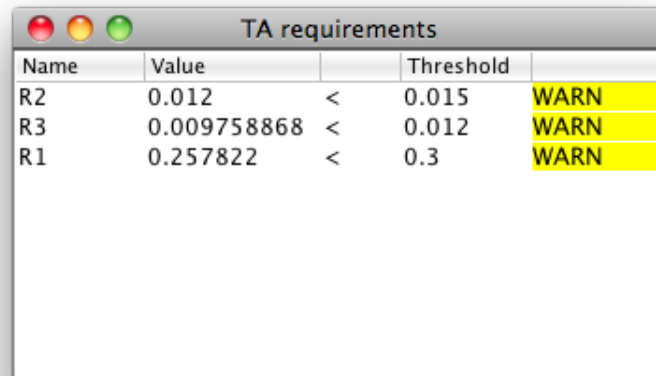
Listing 5.1: Sistema TA (TeleAssistance.dtmc)

La prima parte del sistema è dedicata alla definizione delle informazioni base del sistema e dei requisiti R1, R2, R3. Il valore di essi è la traduzione dei requisiti in formule PCTL (Probabilistic Computation Tree Logic) [13] richieste da PRISM per il model checking. Gli attributi *operator* e *threshold* invece consentono di impostare il valore della soglia che la proprietà non funzionale richiesta deve rispettare, definita dal progettatore del sistema.

Nella seconda parte viene formalizzata la struttura del modello, definendo gli stati del sistema e le transizioni.

Dopo aver caricato il sistema TA, il DTMC model plugin effettua una prima analisi del modello, verificando i requisiti utilizzando i valori caricati dal sistema. I requisiti, con i valori dei parametri calcolati da PRISM, sono visualizzati attraverso una finestra del plugin, come visualizzato in Figura 5.5. Il modello iniziale quindi rispetta tutti i requisiti, sebbene con valori molto vicino alla soglia. Il DTMC avvisa attraverso un messaggio di colore giallo che la differenza tra il valore del requisito e la soglia è minore del 20% della soglia stessa.

Terminata questa fase iniziale, il DTMC model plugin attende la pubblicazione degli eventi provenienti dal nodo KAMI monitor e procede con l'aggiornamento del modello. A seconda della quantità di dati che si ipotizza di ricevere mediamente, è importante impostare correttamente il parametro che regola la frequenza di verifica del mo-



Name	Value		Threshold	
R2	0.012	<	0.015	WARN
R3	0.009758868	<	0.012	WARN
R1	0.257822	<	0.3	WARN

Figura 5.5: Requisiti del modello TA calcolati dal DTMC model plugin

dello, attraverso PRISM. Questo perché l'operazione di model checking è abbastanza onerosa in termini di tempo, in quanto PRISM viene eseguito in un processo separato e l'elaborazione impiega mediamente 3-4 secondi.

L'elaborazione a tempo di esecuzione dei dati del sistema TA permette quindi al model plugin di raffinare le probabilità di transizione tra le configurazioni del sistema e di verificare periodicamente i requisiti. La finestra grafica del DTMC model plugin visualizza in tempo reale i valori aggiornati dei requisiti.

Appena avviene un'infrazione dei requisiti, il model plugin pubblica un evento di violazione. Il plugin in esecuzione nel nodo KAMI reactor è sottoscritto a questo tipo di messaggi: memorizza quindi le informazioni del requisito che è stato violato e, se è stato raggiunto un numero di volte prefissato da un parametro del plugin, attiva azioni di riconfigurazione per far rientrare la violazione e riportare il valore del parametro entro il limite richiesto dal requisito.

Capitolo 6

Conclusioni e sviluppi futuri

In questa tesi è stata presentata una metodologia che permette di mantenere modelli software aggiornati a tempo di esecuzione, concentrandosi maggiormente su requisiti non funzionali di tipo probabilistico come affidabilità e prestazioni. Utilizzando i dati estratti dalle istanze del sistema in esecuzione, i modelli vengono continuamente aggiornati, migliorando la rappresentazione del sistema e verificando periodicamente i requisiti richiesti.

È stato proposto un framework distribuito che supporta l'approccio descritto, effettuando analisi in tempo reale. La struttura di KAMI è basata su plugin, permettendo di interfacciarsi facilmente con i sistemi esistenti. La funzionalità di hot deploy aggiunge la possibilità di operare in un contesto dinamico, dove modelli e sistemi possono cambiare parametri, requisiti e struttura durante l'esecuzione. L'integrazione con il middleware publish-subscribe REDS estende in ambito distribuito le potenzialità del framework, permettendo di utilizzare diverse istanze di KAMI, ognuna con un differente ruolo, per effettuare molteplici analisi.

Infine la metodologia è stata descritta attraverso un caso di studio, verificando le funzionalità dei componenti sviluppati e descrivendo l'approccio in uno scenario concreto. Costruendo il modello DTMC del sistema, è stato possibile calcolare i requisiti di affidabilità richiesti e monitorarli in tempo reale durante l'esecuzione. Monitoraggio del sistema, analisi del modello e gestione delle riconfigurazioni sono stati distribuiti su tre nodi differenti, separando completamente i tre livelli di computazione.

Attualmente, l'approccio descritto prevede la definizione del mo-

dello e dei requisiti nello stesso file. Questo comporta una dipendenza fisica (legata al file XML in cui sono specificati) dei requisiti dal modello, perché per poter modificare i requisiti è necessario caricare di nuovo tutto il modello e far partire una nuova istanza del model plugin associato. È quindi opportuno considerare una modifica della metodologia che prevede un ulteriore disaccoppiamento tra modello e requisiti.

L'utilizzo del model checker probabilistico PRISM introduce un problema relativo al tempo di analisi del DTMC model plugin. Il model checking viene effettuato in tempi troppo lunghi, causando lunghe code nel caso di modelli che ricevono dati ad una frequenza medio-alta. Per evitare il problema, è fondamentale impostare il parametro che rappresenta la frequenza con cui PRISM viene invocato, oppure ricorrere a metodi alternativi a PRISM.

Il lavoro di questa tesi è stato svolto con l'obiettivo di supportare gli ingegneri del software durante tutto il processo di sviluppo, per ottenere sistemi affidabili e in continua evoluzione in cui i modelli convivono con l'implementazione, fino a raggiungere l'adattamento a tempo di esecuzione.

Bibliografia

- [1] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, December 2007.
- [2] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issues and challenges. *Computer*, 39(10):36–43, October 2006.
- [3] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. *Proceedings of the 3rd International Conference on Service Oriented Computing*, 2005.
- [4] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience, New York, NY, USA, 1998.
- [5] BPEL. <http://www.oasis-open.org/>.
- [6] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik. Early prediction of software component reliability. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 111–120. ACM, 2008.
- [7] G. Cugola and G. P. Picco. Reds: A reconfigurable dispatching system. In *SEM*, pages 9–16, 2006.
- [8] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 225–234, New York, NY, USA, 2007. ACM.

-
- [9] H Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. *Automated Software Engineering*, 0:152, 2003.
- [10] C.W. Fraser and D.R. Hanson. Mercury LoadRunner Monitor Reference. *Mercury Interactive*, 2004.
- [11] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM.
- [12] S. S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Trans. Dependable Sec. Comput.*, 4(1):32–40, 2007.
- [13] H Hans and J. Bengt. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.
- [14] JMeter. <http://jakarta.apache.org/jmeter/>.
- [15] S. Nakajima. Model-checking verification for reliable web service. *OOPSLA 2002 Workshop on Object-Oriented Web Services, Seattle, Washington*, 2002.
- [16] PRISM. <http://www.prismmodelchecker.org/>.
- [17] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.
- [18] F. Raimondi, J. Skene, L. Chen, and W. Emmerich. Efficient Monitoring of Web Service SLAs. *UCL, Dept. of Computer Science. Research Note RN/07/01. Gower St, London WC1E 6BT, UK*, 2007, to appear at FSE 2008.
- [19] N. Sato and K. S. Trivedi. Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 107–118, Berlin, Heidelberg, 2007. Springer-Verlag.

-
- [20] C. U. Smith and L. G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.
- [21] C. C. Strelhoff, J. P. Crutchfield, and A. W. Hübler. Inferring markov chains: Bayesian estimation, model comparison, entropy rate, and out-of-class modeling. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 76(1), 2007.
- [22] T. Zheng, M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, 2008.