Capitolo 5

Implementazione

L'architettura software del progetto sviluppato è composta da tre moduli principali:

1. il compilatore offline, che comprende un compilatore per il linguaggio di specifica e il preprocessing dei pattern space;
2. la macchina diagnostica greedy, che rappresenta il motore diagnostico greedy;
3. la macchina diagnostica lazy, che attua la tecnica di ricostruzione lazy.

La specifica dell'istanza di un sistema attivo complesso di cui si vuole determinare le diagnosi è espressa tramite un linguaggio creato ad hoc, la cui grammatica è fornita nell'appendice A. Un esempio di specifica, relativo al problema diagnostico 4.8, è riportato nell'appendice B. Durante la lettura del file di specifica, sono generate le classi opportune e sono effettuati i controlli del caso. Se questa procedura di compilazione non contiene errori (lessicali, sintattici o semantici), viene effettuato il preprocessing offline del problema, durante il quale si creano gli automi relativi ai pattern space. Tutte le informazioni ottenute vengono salvate su appositi file, in modo che non sia necessario dover effettuare nuovamente la procedura in fase di lavoro online, ovvero quando viene avviata la macchina diagnostica, sia essa in modalità greedy o lazy. La procedura diagnostica legge i file generati da una precedente compilazione della specifica e genera le soluzioni candidate. Il linguaggio di programmazione adottato è il C++, che è stato scelto in quanto buon compromesso tra le esigenze di efficienza richieste dal problema e il livello di astrazione tipico della programmazione ad oggetti.

5.1 Rappresentazione degli automi

Per la rappresentazione degli automi è stata ricercata una libreria esterna che fornisse da un lato delle strutture dati e dei metodi efficienti, dall'altro che fosse quanto più generica e personalizzabile, dal momento che gli automi necessari alle tecniche risolutive possiedono vari tipi di informazione negli stati e nelle transizioni. La libreria ASTL(*Automata*

Standard Template Library)[9] soddisfa entrambe le esigenze. Le due classi principali di questa libreria utilizzate nell'implementazione sono:

- `DFA_map<SIGMA,TAG>` rappresenta automi deterministici ed è implementata memorizzando le transizioni di uno stato in uno standard `map`, associando i relativi simboli dell'alfabeto agli stati raggiunti. Dato che la struttura `map` costituisce un albero binario, le operazioni di accesso, inserimento e cancellazione sono effettuate in un tempo logaritmico $\mathcal{O}(\log n)$ rispetto al numero di elementi contenuti. Si tratta di un buon compromesso tra tempi richiesti dalle operazioni e memoria occupata.

- `NFA_mmap<SIGMA,TAG>`, rappresenta automi non deterministici. Le transizioni sono memorizzate per mezzo di `multimap`, le cui principali operazioni hanno complessità logaritmica.

Il template `SIGMA` permette di specificare l'alfabeto cui i simboli di transizione appartengono, mentre il template `TAG` consente di associare ad uno stato delle informazioni satelliti generiche. Per la visualizzazione degli automi, in fase di debug nonché per le illustrazioni di questa trattazione, sono stati utilizzati metodi che generano file scritti nel linguaggio `dot`, interpretati dal programma *Graphviz*[11] che ne genera la rappresentazione.

5.2 Classi condivise

I tre moduli che compongono l'architettura software ideata hanno delle classi tra loro condivise, in modo da operare sulle stesse strutture dati. Si tratta di quegli oggetti che sono creati in fase di compilazione del file di specifica e che successivamente sono utilizzati al fine di avere le informazioni necessarie nella fase diagnostica. Nel seguito vengono descritte tali classi.

- `ComponentModel`
  Rappresenta il modello di un componente. Ha come attributi la stringa del nome che lo identifica e vettori di stringhe contenenti i nomi di eventi, terminali di input, terminali di output e stati. Possiede un vettore di oggetti `Transition` che contengono le informazioni relative ad ogni transizione del modello. L'attributo `automaton` indica l'automa risultante dal modello comportamentale descritto dalle transizioni. Si noti che l'automa non è provvisto di stato iniziale, in quanto questa informazione è definitiva solo nella specifica del problema diagnostico.

- `NetComponent`
  Rappresenta un componente generico nel modello di un nodo. Ha come attributi il nome, che lo identifica nel particolare modello di nodo nel quale è contenuto, e un riferimento ad un oggetto `ComponentModel` che ne rappresenta il modello topologico e comportamentale.

- `Component`
  Rappresenta un componente concreto del problema diagnostico. É una classe derivata da `NetComponent` e possiede gli attributi aggiuntivi relativi all'automa ottenuto dal modello di componente associato, con l'informazione addizionale dello stato iniziale, e gli oggetti `Terminal` riferiti ai terminali di input e output della sua topologia.

- `NetworkModel`
  Rappresenta il modello di un nodo. Possiede gli attributi seguenti:

  - `name`: nome che identifica il modello del nodo;
  - `components`: vettore di oggetti `NetComponents` contenuti nel modello;
  - `inputs` e `outputs`: specificano i nomi dei terminali di input e dei terminali di output del modello del nodo;
  - `links`: contiene l'elenco dei link che connettono tra loro i componenti e i terminali di input del nodo ai terminali di input dei componenti che gestiscono i pattern event in ingresso;
  - `patterns`: vettore di oggetti di tipo `Pattern`, ognuno dei quali rappresenta una dichiarazione di un pattern event che è generato in corrispondenza di un terminale di output del nodo;
  - `initials`: nomi degli stati iniziali per ogni componente del nodo;
  - `viewer`: `map` tra nomi di transizioni di componenti e corrispondenti label osservabili;
  - `ruler`: `map` tra nomi di transizioni di componenti e corrispondenti label di guasto;
  - `pattern_space`: vettore di automi che costituiscono i pattern space del nodo;
  - `languages`: vettore di linguaggi, ognuno dei quali costituito da un insieme di stringhe recanti i nomi delle transizioni appartenenti al linguaggio di una o più dichiarazioni di pattern.

  Si noti che le informazioni relative agli stati iniziali, al viewer e al ruler possono non essere presenti oppure venire sovrascritte nelle successive dichiarazioni di istanza del nodo del sistema o del problema diagnostico.

- `Pattern`
  Rappresenta una dichiarazione di pattern e contiene gli attributi relativi al nome del pattern event, all'espressione regolare, al nome del terminale di destinazione del pattern event, un attributo booleano che indica se il linguaggio associato al pattern è massimo, ovvero costituito dall'insieme di tutte le transizioni di tutti i componenti del nodo, e un vettore di elementi del linguaggio.

- `Terminal`
  rappresenta un terminale di un componente o di un nodo. É caratterizzato da una stringa del nome, una relativa al valore contenuto (evento), e un vettore di riferimenti a terminali ad esso connessi.

- `Transition`
  Rappresenta la transizione di un modello di componente. Ha come attributi la stringa relativa al suo nome identificativo, una coppia di stringhe indicanti il nome dell'evento in ingresso e il terminale di input, un vettore di coppie di stringhe indicanti il nome dell'evento di uscita e il terminale di output, e la coppia del nome di stati sorgente-destinazione coinvolti nella transizione. Possiede attributi e metodi (l'operatore di confronto e di uguaglianza) in modo da permetterne l'utilizzo come alfabeto di un automa della libreria ASTL, come mostrato in figura 5.1.

- `NetTransition`
  Rappresenta la transizione di un componente del modello di un nodo. Ha come attributi il riferimento alla transizione del modello di componente del particolare componente del nodo, e un riferimento a quest'ultimo. Il nome è ottenuto, per mezzo del costruttore dell'oggetto, concatenando il nome della transizione associata e, tra parentesi, il nome del componente del nodo associato.

- `SysTransition`
  Rappresenta la transizione di un componente concreto di un nodo del problema diagnostico. É una classe derivata da `NetTransition` e presenta i seguenti attributi aggiuntivi:

  - `node`: il riferimento nodo del sistema relativo alla transizione;
  - `input_event`: una coppia formata dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del sistema (utile per la diagnosi greedy);
  - `output_events`: un vettore di coppie formate dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del sistema (utile per la diagnosi greedy);
  - `lazy_input_event`: una coppia formata dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del nodo coinvolto (utile per la diagnosi lazy);
  - `lazy_output_events`: un vettore di coppie formate dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del nodo coinvolto (utile per la diagnosi lazy).

- `System`
  Rappresenta la specifica del sistema. Possiede i seguenti attributi:
  - `name`: nome del sistema;
  - `node_list`: vettore dei nodi del sistema;
  - `emergence`: lista delle dichiarazioni di link tra nodi del sistema;
  - `dependency_graph`: grafo che rappresenta la topologia del sistema;
  - `acyclic`: attributo booleano di valore `true` se la topologia del sistema è aciclica e permette quindi la diagnosi lazy.

- `SystemNode`
  Rappresenta un nodo del sistema ed è caratterizzato da un nome identificativo e un riferimento all'oggetto di tipo `NetworkModel` che ne costituisce il modello; possiede inoltre gli attributi relativi agli stati iniziali dei suoi componenti, al viewer e al ruler locali, i quali possono essere ereditati dal modello o sovrascritti nella specifica del problema diagnostico.

- `Problem`
  Rappresenta il problema diagnostico e possiede un nome, una lista di nodi del problema e un vettore di indici che indica l'ordinamento topologico dei nodi, al fine di poter effettuare la diagnosi lazy.

- `ProblemNode`
  Rappresenta un nodo del problema diagnostico. Possiede i seguenti attributi:
  - `name`: nome del nodo, ottenuto dal nodo del sistema associato;
  - `ref_node`: riferimento al nodo del sistema;
  - `concrete_components`: vettore di componenti concreti;
  - `input_terminals`: vettore di terminali di ingresso fisici del nodo;
  - `output_terminals`: vettore di terminali di uscita fisici del nodo;
  - `observation`: sequenza di label relative all'osservazione locale;
  - `index_space`: automa lineare ottenuto dall'osservazione, utile per eventuali sviluppi futuri nei quali l'osservazione è incerta;
  - `depends`: lista di indici dei nodi dai quali il nodo corrente dipende topologicamente;
  - `patt_map`: mappa un pattern event nel terminale di uscita fisico del nodo;
  - `patt_indexes_map`: mappa un pattern event nell'indice della tupla dei terminali di input dei componenti dell'intero problema (utile per la diagnosi greedy);

  - `lazy_patt_indexes_map`: mappa un pattern event nell'indice della tupla dei terminali di input dei componenti del nodo (utile per la diagnosi lazy);

  La classe contiene inoltre le informazioni definitive riguardanti gli stati iniziali dei componenti, il viewer e il ruler locali, che possono essere ereditati dalla specifica del nodo del sistema.

- `Utils`, contiene metodi statici di utilità.

```
class Transition : public CHAR_TRAITS<Transition>
{
    ...
    //required definitions to use a Transition
    //as automata alphabet for ASTL lib
    typedef Transition char_type;
    typedef long        int_type;
    static const size_t size;
    static bool eq(const char_type &x, const char_type &y)
    { return x == y; }
    static bool lt(const char_type &x, const char_type &y)
    { return x < y; }
    bool operator<(const Transition t) const {return name<t.name;}
    bool operator==(const Transition t) const {return name == t.name;}
};
```

Figura 5.1: Classe Transition

In figura 5.2 è riportato il diagramma UML delle classi relative ai componenti del sistema, ovvero le classi `ComponentModel`, `NetComponent` e `Component`.
In figura 5.3 è riportato il diagramma UML delle classi relative alle transizioni, ovvero `Transition`, `NetTransition` e `SysTransition`.
In figura 5.4 è riportato il diagramma UML delle classi che rappresentano, nei diversi livelli di definizione, i nodi del sistema, cioè le classi `NetworkModel`, `SystemNode` e `ProblemNode`.

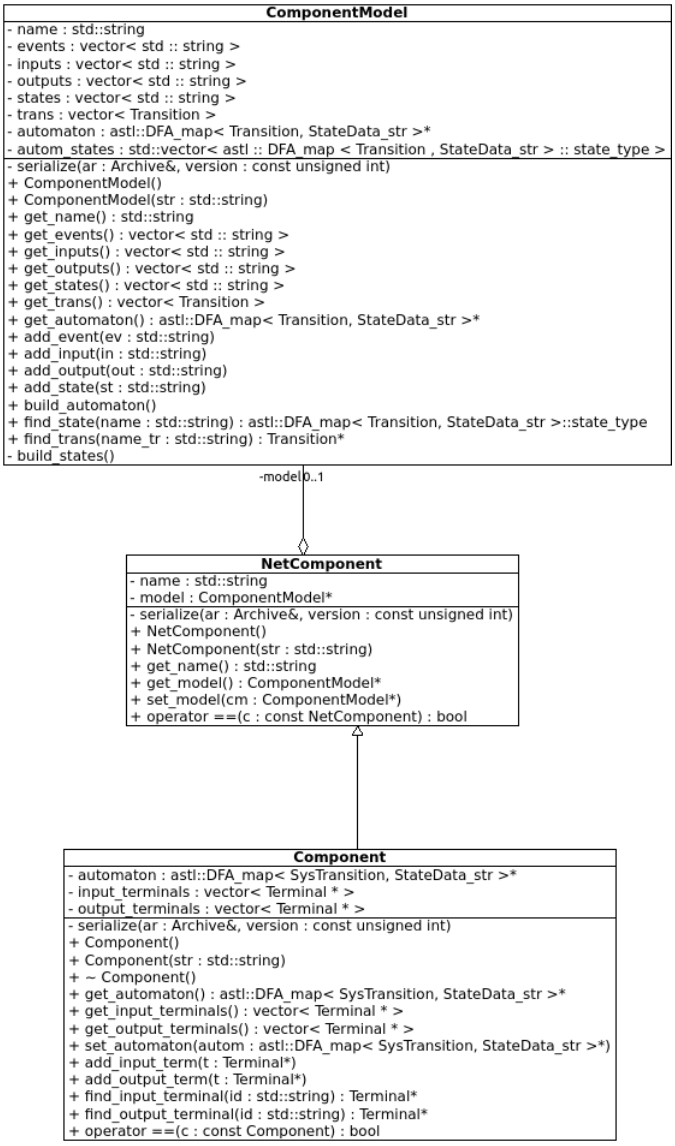


Figura 5.2: Classi dei componenti

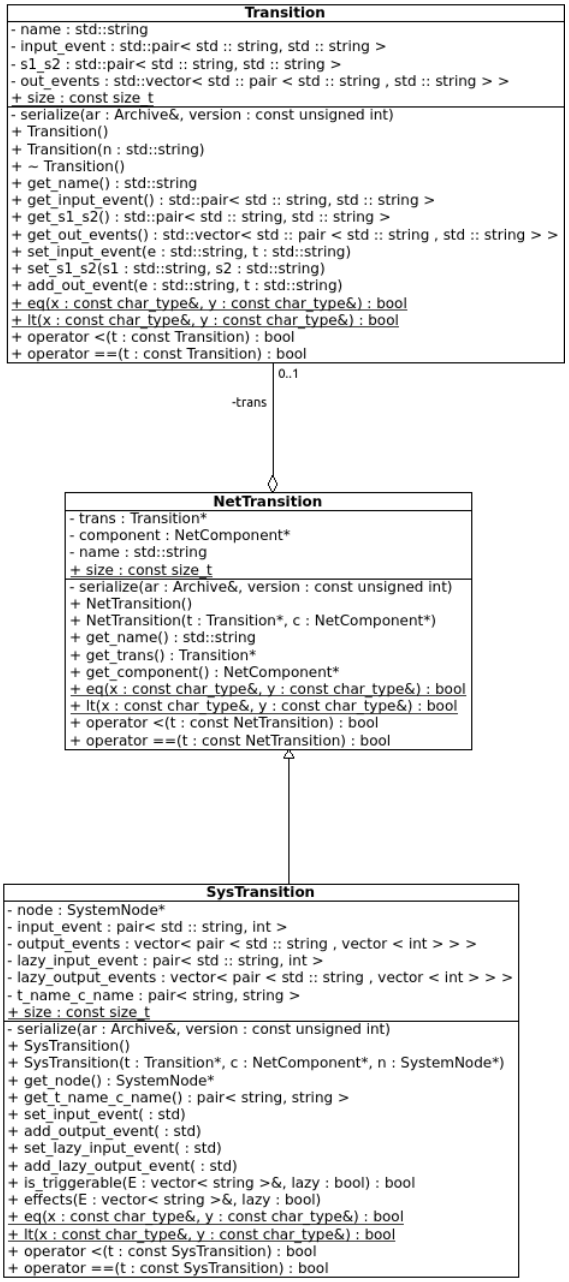


Figura 5.3: Classi delle transizioni

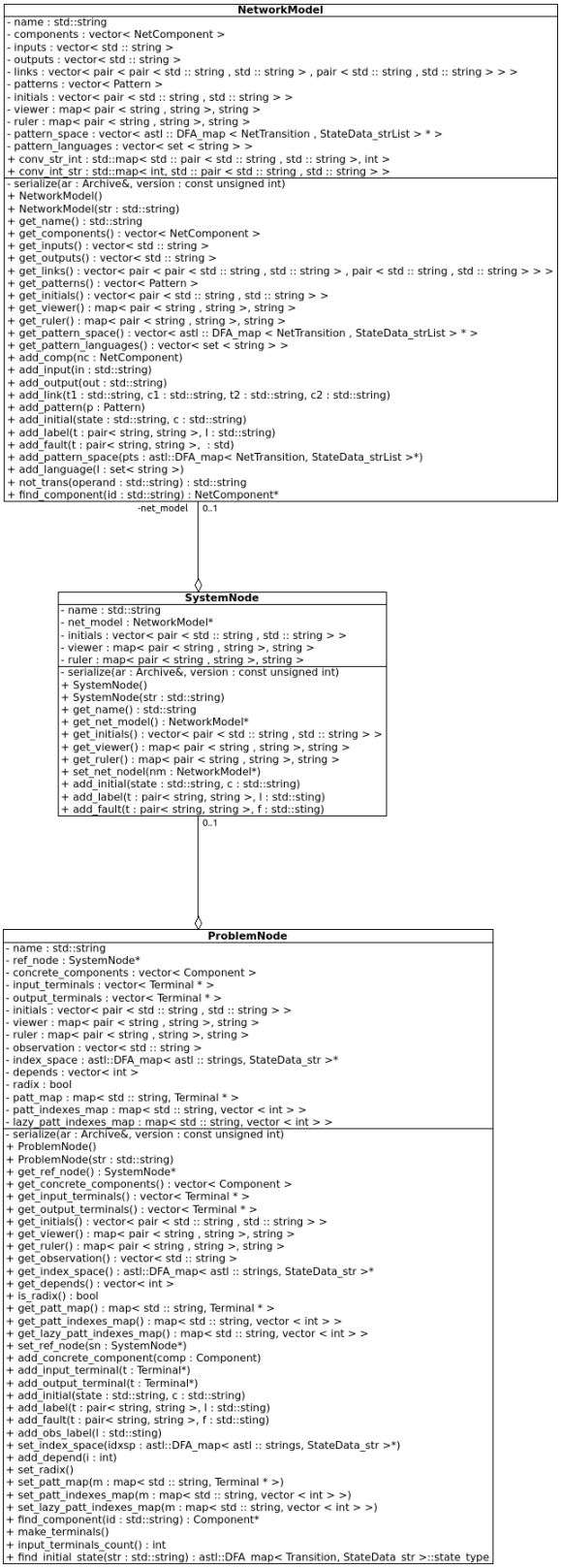


Figura 5.4: Classi dei nodi

5.3 Compilatore offline

Il linguaggio di specifica dei sistemi attivi complessi è fornito, tramite notazione BNF, in appendice A. Il file di specifica è articolato in quattro parti principali:

- modelli dei componenti, a cui i componenti concreti del sistema si riferiscono;
- modelli dei nodi, a cui i nodi concreti del sistema si riferiscono;
- sistema, che istanzia i nodi e definisce i link responsabili dell'invio di pattern event tra di essi;
- problema, che fornisce informazioni necessarie per il problema diagnostico, come le osservazioni locali dei nodi.

I viewer locali, i ruler locali e gli stati iniziali dei componenti dei singoli nodi possono essere definiti, con eventuale riscrittura, a tre diversi livelli:

- nella definizione del modello del nodo;
- nella dichiarazione di un nodo del sistema;
- nella specifica di un nodo del problema.

Questo permette una certa facilità di scrittura, non essendo costretti a dichiarare informazioni di questo tipo, quando esse sono ridondanti. Il file di specifica viene compilato seguendo le tradizionali fasi ideate per i compilatori dei linguaggi di programmazione:

1. analisi lessicale, che legge il flusso di caratteri che costituiscono il codice di specifica e raggruppa i caratteri in sequenze significative dette lessemi, quali le parole chiave del linguaggio e gli identificatori;
2. analisi sintattica, che riceve i lessemi dall'analizzatore lessicale per creare una rappresentazione ad albero del codice di specifica, detto albero sintattico;
3. analisi semantica, la quale utilizza le informazioni raccolte nell'albero sintattico in modo da verificare la consistenza del codice di specifica.

Si noti che in questo progetto è possibile articolare una specifica in più file, esplicitando delle dichiarazioni del tipo `#include` per i file che si vogliono incorporare. Questo permette il riutilizzo di specifiche già dichiarate precedentemente, aumentando inoltre la leggibilità dei file di specifica.

5.3.1 Analizzatore lessicale

L'analizzatore lessicale (o scanner) è implementato mediante lo strumento *Flex*, il quale permette di dichiarare espressioni regolari che descrivono i pattern per i token, ovvero coppie nome-attributo, ad esempio `<id,breaker>`. La struttura di un programma Flex è composta da tre parti principali:

1. dichiarazioni, che includono variabili, costanti e definizioni regolari;
2. regole di traduzione, che specificano per ogni pattern dichiarato per mezzo di una espressione regolare le azioni da compiere, attraverso frammenti di codice;
3. funzioni ausiliarie, che possono essere richiamate nelle azioni dei pattern.

Sebbene Flex abbia un'opzione per creare direttamente uno scanner C++, il codice generato spesso non funziona e contiene errori[4] . La procedura adottata per risolvere questo problema è quella di compilare il programma Flex scritto in linguaggio C per mezzo di un compilatore C++. La sezione dichiarativa del linguaggio è riportata in figura 5.5. In essa si definiscono spazi, delimitatori, commenti di linea (in stile C) e l'espressione regolare che definisce gli identificatori `id`, formata da una lettera a cui seguono altre lettere, numeri oppure il carattere `_`.

```
// The location of the current token.
yy::location loc;

%}
%option noyywrap nounput
%x incl

delimiter [ \t]
spacing {delimiter}+
letter [A-Za-z]
digit [0-9]
unscore         _
id {letter}({letter}|{digit}|{unscore})*
eol \n
comment \/\/.*

%{
  // Code run each time a pattern is matched.
  # define YY_USER_ACTION  loc.columns (yyleng);
%}
```

Figura 5.5: Dichiarazioni.

Le regole di traduzione utilizzate sono principalmente volte a generare i token relativi a parole chiave del linguaggio e agli identificatori. In corrispondenza di direttive `#include`,

il passaggio di lettura al file incluso è operato, come mostrato in figura 5.6, creando una pila di buffer.

```
#include          BEGIN(incl);
<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+
{   /* got the include file name */
    chdir(INPUT_FILE_DIR);
    FILE *file = fopen( yytext, "r" );

    if (file == NULL)
    {
       cout << "Input file error: included file does not exist" << endl;
       exit(1);
    }
    yyin = file;

    yypush_buffer_state(yy_create_buffer( yyin, YY_BUF_SIZE ));
    BEGIN(INITIAL);
}
```

Figura 5.6: Gestione delle direttive di inclusione.

Gli spazi e i commenti non invocano alcuna azione, mentre la fine del file attuale toglie dalla pila il buffer corrente, riportando il controllo alla scansione del file precedente da cui la direttiva di inclusione è stata chiamata (figura 5.7). Se il buffer corrente è l'unico elemento della pila, il processo di scanning termina ed è generato il token relativo alla fine del file.

```
{spacing} ;
{comment}   ;

{eol} {loc.lines (yyleng);loc.step();}

<<EOF>>
{
  yypop_buffer_state();
  if ( !YY_CURRENT_BUFFER )
    return yy::spec_parser::make_END_OF_FILE(loc);;

}
```

Figura 5.7: Azioni compiute alla fine del file.

Quando viene trovata una parola chiave (come ad esempio `component` e `model`), viene creato il corrispondente token che verrà utilizzato nell'analisi sintattica. Per quanto riguar-

da gli identificatori (`id`), il loro token è composto oltre che dalla posizione, dalla stringa che ne rappresenta il nome, memorizzato durante lo scanning nella variabile `yytext` (figura 5.8). La lettura di un carattere non appartenente a nessuna delle configurazioni definite genera un errore lessicale.

```
component   return yy::spec_parser::make_COMPONENT(loc);
model       return yy::spec_parser::make_MODEL(loc);
is          return yy::spec_parser::make_IS(loc);


{id}        return yy::spec_parser::make_ID(yytext,loc);
.     {cout << "Lexical error"; exit(1);}
```

Figura 5.8: Regole di traduzione per parole chiave e identificatori.

5.3.2 Analizzatore sintattico

L'analizzatore sintattico (parser) è implementato tramite lo strumento *Bison*. La generazione della grammatica è supportata dalla classe `spec_driver`, la quale avvia il parsing e ne gestisce gli eventuali errori generati. Il file della grammatica inizia richiedendo lo skeleton parser deterministico per poter utilizzare direttamente il linguaggio C++, creando il file header della grammatica e specificando il nome della classe del parser.

```
%skeleton "lalr1.cc"
%defines
%define parser_class_name {spec_parser}
```

Per poter utilizzare oggetti C++ come valori semantici della grammatica, deve essere richiesta l'interfaccia `variant`.

```
%define api.value.type variant
%define api.token.constructor
%define parse.assert
```

Successivamente sono specificate le dichiarazioni necessarie ai valori semantici. Poiché il parser utilizza la classe driver e viceversa, questa mutua dipendenza è risolta fornendo una *forward declaration*.

```
%code requires
{
...
class spec_driver;
}
```

L'oggetto driver viene passato al parser e allo scanner tramite referenza.

```
%param { spec_driver& driver}
```

Quindi viene richiesto di tenere traccia della locazione (riga e colonna del file di specifica), passando il nome del file dell'oggetto `driver`.

```
%locations
%initial-action
{
  // Initialize the initial location.
  @$.begin.filename = @$.end.filename = &driver.file;
};
```

La definizione dei token avviene come di seguito.

```
%token
COMPONENT
MODEL
IS
...
END_OF_FILE  0  "end of file"
COMMA ","
...
```

Per ogni terminale e non terminale della grammatica, deve essere specificato un tipo dell'oggetto.

```
%token <string> ID "id"
%type <ComponentModel>  comp_model_decl
%type <vector<string> > event_decl
..
```

Successivamente viene implementata la grammatica del linguaggio, assegnando i valori semantici definiti.

5.3.3 Analizzatore semantico

L'analisi semantica, a differenza di quella lessicale e sintattica, non prevede l'utilizzo di strumenti standard per la sua attuazione, ma è compito del programmatore definirne le regole, in base al significato del linguaggio specificato Nell'ambito di questo progetto, i controlli semantici sono affidati alla classe `spec_driver` (figura 5.9), la quale verifica la

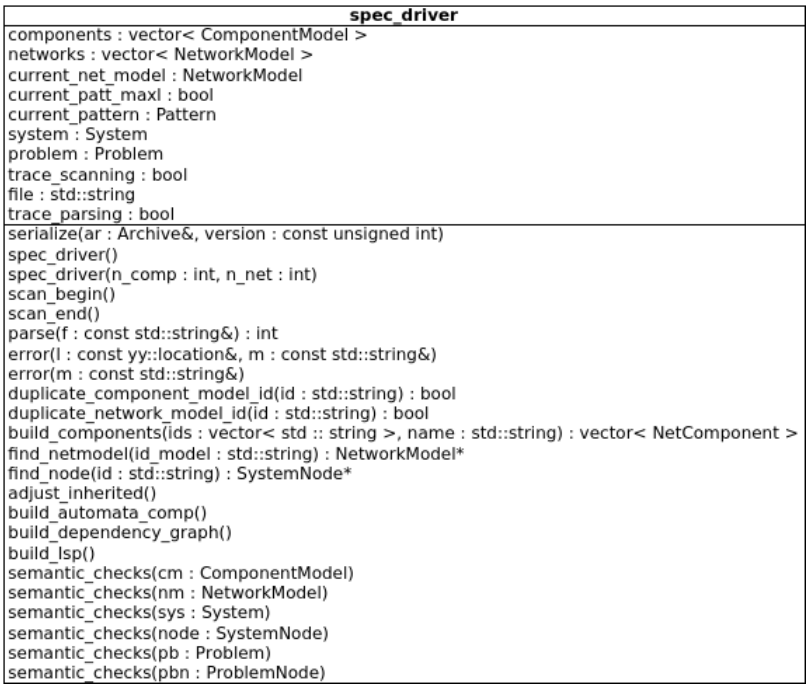


Figura 5.9: Classe specdriver

consistenza semantica durante il parsing del file, interrompendone l'esecuzione in caso di errori.

La classe possiede, tra gli altri, i metodi `semantic_checks`, implementati in *overloading* per i seguenti tipi di parametro d'ingresso:

1. `ComponentModel`: verifica che non vi siano identificatori multipli per le liste di eventi, terminali di input, terminali di output e stati; nelle dichiarazioni delle transizioni, verifica che il nome degli eventi, dei terminali e degli stati siano stati precedentemente definiti.

2. `NetworkModel`: verifica l'unicità del nome dei componenti, dei terminali e dei nomi dei pattern; verifica che i terminali e i componenti dei link siano definiti. Verifica che non vi siano più link entranti nello stesso terminale di input. Verifica l'esistenza delle transizioni utilizzate nei pattern, nel viewer e nel ruler (se presenti), oltre all'esistenza degli eventuali stati iniziali dei componenti dichiarati.

3. `System`: verifica che nelle dichiarazioni di *emergence* i nomi dei nodi e dei relativi terminali siano stati definiti.

4. `SystemNode`: effettua i controlli sulle eventuali dichiarazioni di stati iniziali, viewer e ruler locali.

5. `Problem`: verifica che siano ivi dichiarati tutti i nodi definiti nel sistema.

6. `ProblemNode`: verifica i controlli sulle eventuali dichiarazioni di stati iniziali, viewer e ruler locali. Se esse non sono presenti, vengono ereditate dal nodo di sistema corrispondente. Viene inoltre verificata l'esistenza delle label dell'osservazione locale nel viewer corrente.

L'analisi semantica viene ultimata, attraverso il metodo `build_dependency_graph`, costruendo il grafo che rappresenta la topologia del sistema, verificando che sia connesso. La connessione è verificata nel metodo `Utils::disconnected_graph`, il quale compie un'intera traversata in ampiezza del grafo; se il numero di nodi visitati è inferiore al numero di nodi dichiarati, alcuni di essi non sono connessi. Viene inoltre verificata l'aciclicità del grafo, eliminando uno dopo l'altro i nodi foglia dal grafo, fino a quando esso non risulta essere vuoto. Se ad un certo punto della procedura non vi sono nodi foglia e il grafo non è vuoto, significa che sono presenti ciclicità e non sarà possibile avviare la diagnosi lazy. Nel caso il grafo sia aciclico, viene calcolato l'ordinamento topologico, attraverso un algoritmo simile al precedente.

5.3.4 Generazione dei pattern space

La principale azione compiuta in fase offline, ovvero in un momento antecedente la diagnosi, è la generazione dei pattern space dei modelli di nodi definiti, come descritto nel paragrafo 4.1.4. Il primo passo di questo algoritmo consiste nella costruzione di un automa equivalente all'espressione regolare definita nella dichiarazione di pattern. Un metodo noto consiste nella costruzione di Thompson descritta nel paragrafo 2.5.1. Per questo scopo, è stata utilizzata la libreria *Grail+*[8], una libreria C++ che permette di definire espressioni regolari e convertirle in automi con un linguaggio equivalente. La conversione avviene utilizzando la sintassi presentata in figura 5.10.

```
fm<int> patodfa(std::string regex)
{
    re<int> r;
    istringstream str(regex);
    str >> r;
    fm<int> dfa;
    r.retofm(dfa);
    dfa.subset();

    return dfa;
}
```

Figura 5.10: Passaggio da espressione regolare ad automa

Data una stringa in ingresso, la funzione implementata inserisce tale stringa in un oggetto della classe `re`(*regular expression*), parametrizzato per mezzo di un template che

indica l'alfabeto di appartenenza dei simboli dell'espressione. Dato che gli alfabeti possibili per questa libreria erano esclusivamente di tipo `char` o `int`, è stato scelto quest'ultimo (poiché 256 simboli possibili potevano non essere sufficienti in applicazioni di dimensioni ragguardevoli), mantenendo una corrispondenza biunivoca tra ogni intero e una transizione, coinvolta nella dichiarazione del pattern, relativa ad un componente del nodo. La conversione da espressione regolare ad automa a stati finiti avviene applicando all'oggetto che rappresenta l'espressione regolare il metodo `retofm`, che crea l'automa nell'oggetto della classe `fm`(*finite machine*) passato come parametro. A seguito della conversione, l'automa è determinizzato applicando il metodo `subset`, il quale esegue la subset construction descritta nel paragrafo2.3. Successive manipolazioni dei diversi automi ottenuti dai pattern dichiarati permettono di unirli, aggiungere le ϵ-transizioni opportune e determinizzare il risultato. L'automa finale viene poi convertito in un automa equivalente, utilizzando le strutture dati della libreria ASTL. Infatti, la libreria Grail+, sebbene utile poiché implementa la costruzione di Thompson, non sembrava indicata ad un utilizzo negli algoritmi di diagnosi, in quanto non permette estensioni per quanto riguarda i dati satellite associati a transizioni e stati dell'automa. La minimizzazione finale del pattern space è implementata nel metodo `Utils::minimize_by_partition`, che esegue una minimizzazione tenendo conto dei tag associati (pattern event) ai nodi finali dell'automa.

5.3.5 Salvataggio dei dati compilati

Una volta che il file di specifica è stato processato, le classi opportune sono state create e i controlli semantici sono andati a buon fine, la procedura di elaborazione offline termina creando dei file binari contenenti tutte le informazioni utili alla successiva fase diagnostica, ovvero tutti gli oggetti istanziati per il problema specificato. La scrittura di questi dati, chiamata serializzazione, è una procedura piuttosto complessa se implementata manualmente in C++, in quanto deve gestire puntatori ed evitare ridondanza. Per questo è stata adottata la libreria *Serialization*, fornita dalla collezione *Boost*[10]. Essa permette di salvare un oggetto con semplici modifiche della classe a cui esso appartiene. Ad esempio, la definizione della classe `ComponentModel` è stata modificata aggiungendo le istruzioni riportate in figura 5.11. Per ogni attributo della classe, viene specificata la lettura e la scrittura in un oggetto di tipo `Archive`, appartenente alla libreria. É necessario esplicitare l'operazione per ogni attributo della classe, in modo da poter salvare lo stato dell'oggetto corrente; eventuali attributi non salvati avranno un valore impredicibile. Quando l'oggetto corrisponde ad un archivio di output, l'operatore `&` indica la scrittura, mentre nel caso di un archivio di input esso indica l'operazione di lettura. Le modifiche attuate permettono quindi sia la scrittura su file che la lettura operata durante la successiva fase diagnostica.

Il programma sviluppato salva le informazioni precompilate in quattro file distinti:

- un file contenente tutti i modelli dei componenti (`component_models.dat`);

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>

class ComponentModel{
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        ar & autom_states;
        ar & name;
        ar & automaton;
        ar & events;
        ar & inputs;
        ar & outputs;
        ar & states;
        ar & trans;
    }
```

Figura 5.11: Modifiche apportate alla classe ComponentModel per la serializzazione.

- un file che comprende tutti i modelli dei nodi (`network_models.dat`);
- un file che racchiude la specifica del sistema (`system.dat`);
- un file contenente le informazioni del problema diagnostico con le osservazioni temporali (`problem.dat`).

Per eseguire la scrittura di un oggetto è sufficiente dare istruzioni simili a quelle riportate in figura 5.12, nella quale è salvato su file il `vector` dei modelli dei componenti utilizzati nel sistema.

```
{
    std::ofstream ofs("../../CompiledData/component_models.dat");
    boost::archive::text_oarchive oa(ofs);
    // write class instance to archive
    oa << driver.components;
    // archive and stream closed when destructors are called
}
```

Figura 5.12: Serializzazione.

Si noti che la libreria *Serialization* permette di salvare, senza il bisogno di istruzioni aggiuntive, oggetti appartenenti alla Standard Template Library, quali ad esempio `vector`, `map`, `list` e `set`. Per quanto riguarda il salvataggio di puntatori, al fine di evitare ridondanza, la libreria salva l'oggetto puntato un'unica volta, mantenendone i riferimenti.

5.4 Macchina diagnostica

5.4.1 Lettura dei dati precompilati

La fase diagnostica necessita degli oggetti generati dal precedente passo di compilazione dei file di specifica. La lettura delle informazioni salvate avviene in modo simile a quanto fatto per la scrittura, come si può vedere in figura 5.13.

```
vector<ComponentModel> components;
{
    // create and open an archive for input
    std::ifstream ifs("../../CompiledData/component_models.dat");
    boost::archive::text_iarchive ia(ifs);
    // read class state from archive
    ia >> components;;
    // archive and stream closed when destructors are called
}
```

Figura 5.13: Lettura dai file binari e memorizzazione degli oggetti salvati.

5.4.2 Modalità greedy

La modalità greedy implementa l'algoritmo di ricostruzione del behavior dell'intero sistema. Dopo aver acquisito gli oggetti memorizzati nei file precompilati, viene generato lo stato iniziale con i seguenti campi:

- una tupla di interi indicanti gli stati iniziali di tutti i componenti del problema;
- una tupla di stringhe recanti il valore, inizialmente vuoto, del contenuto dei terminali di input di tutti i componenti del sistema;
- una tupla di interi indicanti gli stati iniziali di tutti i pattern space di ogni nodo;
- una tupla di interi indicanti gli stati iniziali degli index space di ogni nodo (osservazione locale, inizialmente nulla).

La classe che rappresenta uno stato del behavior è riportata in figura 5.14.

L'algoritmo tiene traccia delle configurazioni di ogni stato attraverso una tabella hash: in questo modo la verifica dell'esistenza di uno stato generato da una transizione avviene in tempo costante. La tabella mappa la stringa relativa al contenuto dello stato nel valore intero associato allo stato del behavior, come mostrato di in figura 5.15 per lo stato iniziale.

La creazione del behavior avviene come descritto nella sezione 4.3. La verifica dell'esistenza di uno stato raggiunto da una transizione avviene come descritto in figura 5.16, dove `tag_s1` è un'istanza della classe `BehaviorState`.

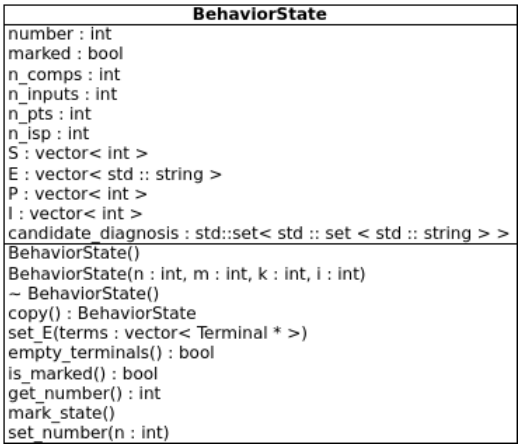


Figura 5.14: Classe BehaviorState

```
DFA_map<SysTransition,BehaviorState> behavior;
unordered_map<string,unsigned int> hash_values;
stringstream ss;
ss << tag_s0;
unsigned int s0 = behavior.new_state();
behavior.tag(s0) = tag_s0;
behavior.initial(s0);
hash_values[ss.str()] = s0;
```

Figura 5.15: Creazione tabella hash

```
stringstream s_str;
s_str << tag_s1;
string str = s_str.str();
unsigned int s1;
try{ s1 = hash_values.at(str);}
catch (const std::out_of_range&)
{
    s1 = behavior.new_state();
    behavior.tag(s1) = tag_s1;
    hash_values[str] = s1;
...
```

Figura 5.16: Ricerca nella tabella hash

Per ogni stato creato viene contestualmente verificato se si tratta di uno stato finale, ovvero caratterizzato da terminali di input vuoti e che ha consumato tutte le osservazioni locali dei vari nodi. Una volta terminato il ciclo principale della generazione del behavior, ed ottenuto quindi il behavior spurio, esso viene potato di tutte le transizioni e gli stati che non sono in un cammino tra lo stato iniziale ed uno stato finale. L'operazione avviene invocando la funzione predisposta `trim` presente nella libreria ASTL (figura 5.17).

```
DFA_map<SysTransition,BehaviorState> bhv;
unsigned int init = trim(bhv,dfirst_markc(behavior));
```

Figura 5.17: Rimozione della parte spuria del behavior

Se dopo questa operazione il behavior risulta vuoto, significa che le osservazioni date non sono consistenti con il modello specificato e l'esecuzione viene terminata con errore. Questo può avvenire in fase di sperimentazione, a causa dell'assenza di un reale sistema fisico che dia un'osservazione vera del comportamento. Altrimenti, l'algoritmo procede decorando l'automa ottenuto e distillandone le diagnosi. L'algoritmo di decorazione consiste nella semplice implementazione dello pseudocodice 3. La distillazione consiste nell'unione delle diagnosi contenute negli stati finali del behavior.

5.4.3 Modalità lazy

La diagnosi lazy avviene in maniera simile alla ricostruzione greedy, con alcune variazioni. Viene letto l'ordinamento topologico del sistema e viene applicata in successione la ricostruzione del comportamento del singolo nodo. Ogni nodo che non è l'ultimo dell'ordinamento topologico, ovvero che non costituisce un nodo radice, necessita della generazione della sua interfaccia. Durante la generazione dei behavior, si tiene conto anche delle dipendenze dalle interfacce dei nodi inferiori collegati al nodo corrente.

Determinizzazione dell'interfaccia

La classe `Determinization` fornisce dei metodi statici che permettono di ottenere l'interfaccia associata al behavior locale di un nodo. É composta dai metodi seguenti:

- `NFAtoDFA` guida il processo di determinizzazione, ricevendo il behavior del nodo;
- `eps_closure` calcola la ϵ-chiusura di uno stato o di un insieme di stati del behavior, nel quale le transizioni che non producono pattern event sono sostituite da ϵ-transizioni;
- `extract_subNFA` estrae dall'automa del behavior la parte contenente gli stati appartenenti alla ϵ-chiusura, in modo che tale sottoautoma possa essere memorizzato nello stato dell'interfaccia;

- `move` calcola gli stati raggiungibili attraverso transizioni, applicate a stati interni allo stato dell'interfaccia, che generano pattern event.

La decorazione del sottoautoma contenuto negli stati dell'interfaccia avviene richiamando un metodo della classe `Decoration` che implementa la classica decorazione come nell'algoritmo 3. La classe che rappresenta lo stato dell'interfaccia è visualizzata in figura 5.18. Le transizioni dell'interfaccia (figura 5.19) possiedono invece i seguenti attributi:

- `trans`: transizione di un componente del nodo;
- `delta`: diagnosi associata alla transizione;
- `pattern_events`: elenco dei patter event associati alla transizione dell'interfaccia.

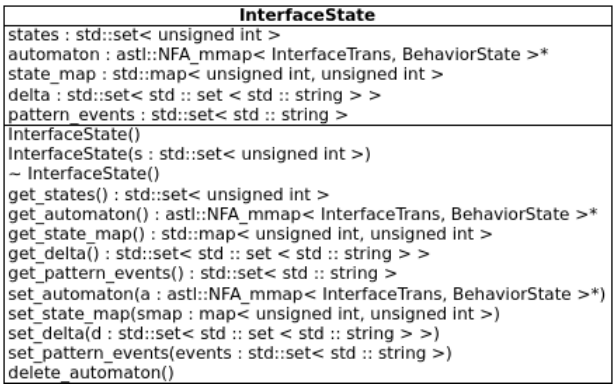


Figura 5.18: Classe InterfaceState

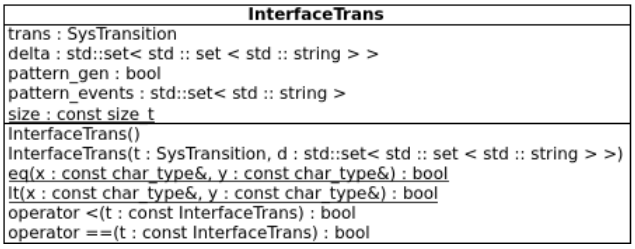


Figura 5.19: Classe InterfaceTrans

Decorazione del behavior del nodo radice

La decorazione del behavior del nodo radice avviene attraverso il metodo `decorate_lazy_bhv` della classe `Decoration`. La procedura consiste in alcune variazioni dell'algoritmo di decorazione della diagnosi greedy:

- una transizione del behavior, se relativa ad una interfaccia, contiene una diagnosi che deve essere propagata agli stati successivi;

- in corrispondenza di uno stato finale del behavior, la decorazione deve tenere conto delle diagnosi relative agli stati finali delle interfacce da cui il nodo corrente dipende.

Capitolo 6

Sperimentazione

La fase di sperimentazione è stata orientata in due principali direzioni:

- confrontare i metodi diagnostici greedy e lazy in termini di risorse computazionali utilizzate;
- analizzare i tempi di esecuzione e la memoria allocata al crescere delle dimensioni delle istanze utilizzate.

L'aspettativa riguardante l'inefficienza del metodo greedy è stata immediatamente confermata, in quanto problemi di diagnosi relativi a sistemi attivi complessi caratterizzati da un numero limitato di componenti esauriscono con estrema facilità la memoria del calcolatore. Dal momento che, come previsto, il metodo lazy è molto più efficiente e permette il calcolo delle diagnosi di sistemi molto più grandi, un'analisi di scalabilità è stata compiuta ristrettamente alla macchina diagnostica lazy. I risultati ottenuti sono estremamente soddisfacenti, in quanto hanno permesso di catturare un andamento lineare nel numero di componenti sia in termini di tempo computazionale, sia in termini di memoria occupata. Gli esperimenti svolti sono stati condotti utilizzando un calcolatore con processore *Intel Core i5* da 2.40GHz, 4GB di RAM, su sistema operativo *Linux Ubuntu 15.10*. Le istanze adottate sono caratterizzate da componenti che possiedono i modelli descritti nelle figure 3.1, 3.2 e 4.3 rispettivamente per la protezione, il breaker e la linea. Di fatto i SAC utilizzati negli esperimenti sono composizioni e variazioni di insiemi di reti elettriche connesse.

6.1 Confronto dei metodi greedy e lazy

Il confronto tra il metodo greedy e quello lazy ha prodotto i risultati riportati in tabella 6.1. Sono state utilizzate 22 istanze di sistemi attivi complessi di dimensioni che vanno da 2 a 20 componenti totali, e un numero di nodi compreso tra 1 e 9. Per le istanze piccole, come immaginabile, la differenza tra i metodi non è significativa. Il comportamento

esponenziale del metodo greedy, tuttavia, emerge chiaramente con la crescita del numero dei componenti, portando alla saturazione della memoria con 20 componenti. I risultati relativi all'ultima istanza del metodo greedy non sono infatti riportati: la saturazione della memoria causa lo swap compiuto da parte del sistema operativo e il calcolo non sembra poter terminare in un tempo accettabile. Il metodo lazy, dal canto suo, si mantiene contenuto sia nel tempo che nella memoria allocata. Le variazioni di prestazioni nell'ambito di quest'ultimo metodo sono dovute alla variazione di dimensione dei nodi del sistema.

		Greedy		Lazy	
Componenti	Nodi	Tempo	Memoria	Tempo	Memoria
2	1	1,39 ms	4,2 MB	1,03 ms	4,3 MB
3	1	4,95 ms	4,2 MB	4,64 ms	4,4 MB
3	2	1,98 ms	4,3 MB	2,45 ms	4,4 MB
4	2	4,41 ms	4,3 MB	4,69 ms	4,4 MB
5	3	4,93 ms	4,8 MB	6,45 ms	4,6 MB
6	3	19,84 ms	4,9 MB	8,90 ms	4,9 MB
7	4	23,98 ms	5,5 MB	6,38 ms	5,2 MB
8	3	81,16 ms	5,8 MB	10,82 ms	5,1 MB
9	4	96,82 ms	6,9 MB	16,52 ms	5,9 MB
9	4	0,13 s	6,6 MB	12,89 ms	5,0 MB
10	5	0,19 s	11,9 MB	19,35 ms	9,4 MB
11	6	0,45 s	12,9 MB	9,02 ms	5,8 MB
12	5	1,23 s	25,1 MB	49,63 ms	5,9 MB
13	5	2,04 s	40,8 MB	96,69 ms	11,0 MB
13	6	1,77 s	55,5 MB	76,29 ms	28,5 MB
14	7	4,78 s	82,6 MB	13,14 ms	5,8 MB
15	5	24,18 s	395,4 MB	0,33 s	11,9 MB
16	5	26,10 s	417,8 MB	0,27 s	11,0 MB
17	6	39,19 s	616,2 MB	1,56 s	38,1 MB
18	7	1,90 min	1,7 GB	30,67 ms	6,2 MB
19	8	4,50 min	3,3 GB	32,82 ms	10,5 MB
20	9			68,53 ms	9,8 MB

Tabella 6.1: Risultati dei test riguardanti il confronto dei metodi greedy e lazy

6.1.1 Tempo di esecuzione

In figura 6.1 viene analizzata, tramite un diagramma cartesiano, la differenza tra i due metodi diagnostici per quanto riguarda il tempo di calcolo. L'asse delle ascisse indica il numero di componenti dell'istanza, mentre l'asse verticale indica, in scala logaritmica, il tempo di esecuzione della diagnosi. Il metodo lazy, pur con delle variazioni in base all'istanza utilizzata, non mostra l'andamento esponenziale del metodo greedy. Ad esempio, per l'ultima istanza confrontabile di 19 componenti, il metodo greedy impiega più di 4 minuti e mezzo, mentre l'algoritmo lazy giunge a termine in soli 32.82ms. L'istanza

successiva di 20 componenti satura la memoria nel caso greedy, quindi i tempi non sono calcolabili. Per quanto riguarda il metodo lazy, invece, sono sufficienti 68.53ms.

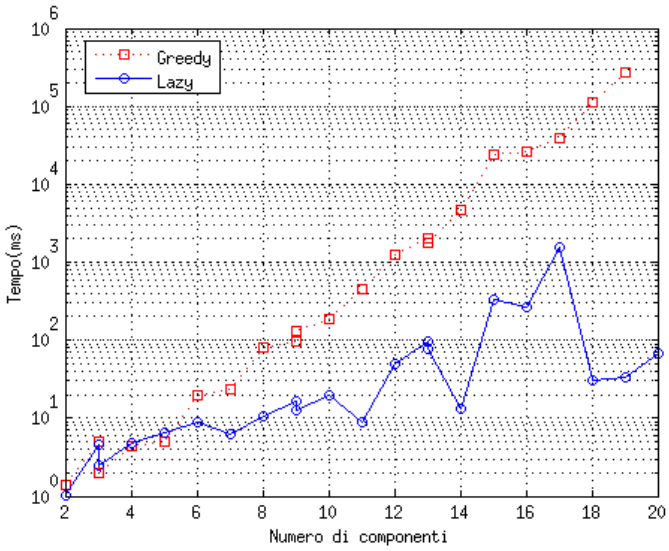


Figura 6.1: Confronto dei tempi di esecuzione tra la diagnosi greedy e lazy

6.1.2 Memoria

In figura 6.2 viene analizzata, tramite un diagramma cartesiano, la differenza tra i due metodi diagnostici per quanto riguarda la memoria allocata. L'asse delle ascisse indica il numero di componenti dell'istanza, mentre l'asse verticale indica, in scala logaritmica, la memoria. Anche in questo caso, il metodo lazy non segue l'andamento esponenziale della diagnosi greedy. Ad esempio, per l'ultima istanza confrontabile di 19 componenti, il metodo greedy occupa 3.3GB, mentre l'algoritmo lazy utilizza solamente 10.5MB. L'istanza successiva di 20 componenti satura la memoria nel caso greedy, mentre con il metodo lazy sono sufficienti meno di 10MB.

6.2 Approfondimento del metodo lazy

Una volta verificata l'impraticabilità della diagnosi greedy, è stata effettuata una serie di esperimenti volti a catturare la complessità dell'algoritmo lazy. In tabella 6.2 sono riportati i tempi di esecuzione e la memoria allocata per istanze di SAC con un numero di componenti che varia da 9 a 638, e un numero di nodi del sistema compreso tra 4 e 255. Si noti che quasi tutte le istanze utilizzate, eccetto le prime due di 9 e 18 componenti, non siano calcolabili attraverso il metodo greedy. Uno stato del behavior del SAC nella sua interezza infatti, potrebbe nel caso peggiore aver un numero di stati, considerando ad esempio l'ultima istanza, esponenziale rispetto ai 638 componenti. Per mezzo del metodo

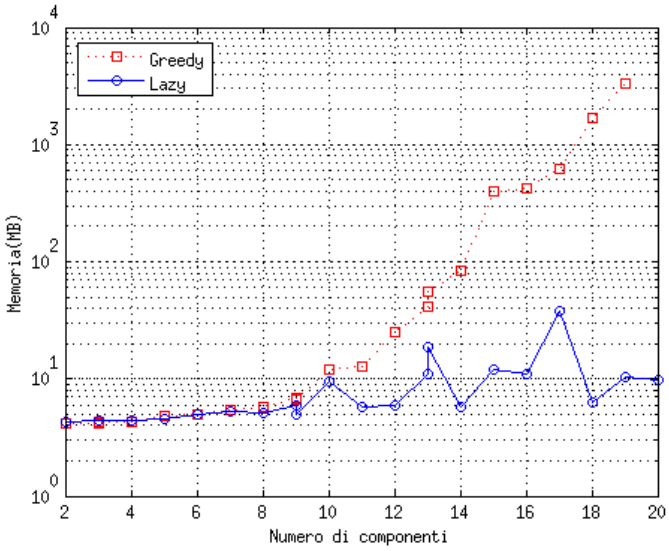
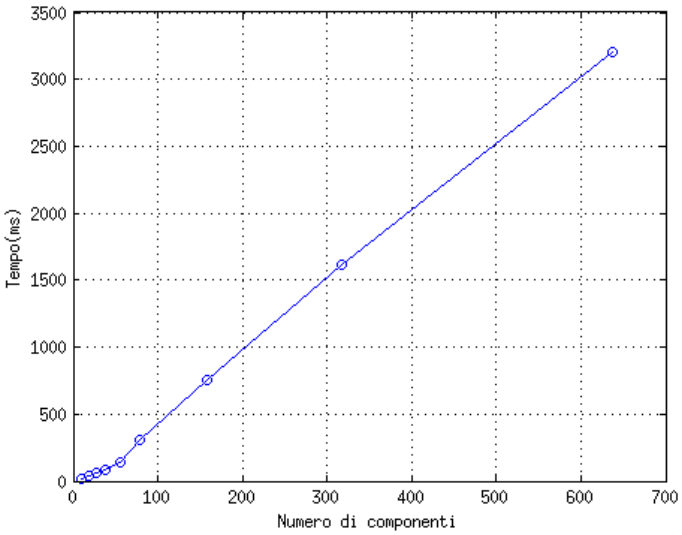


Figura 6.2: Confronto della memoria allocata tra la diagnosi greedy e lazy

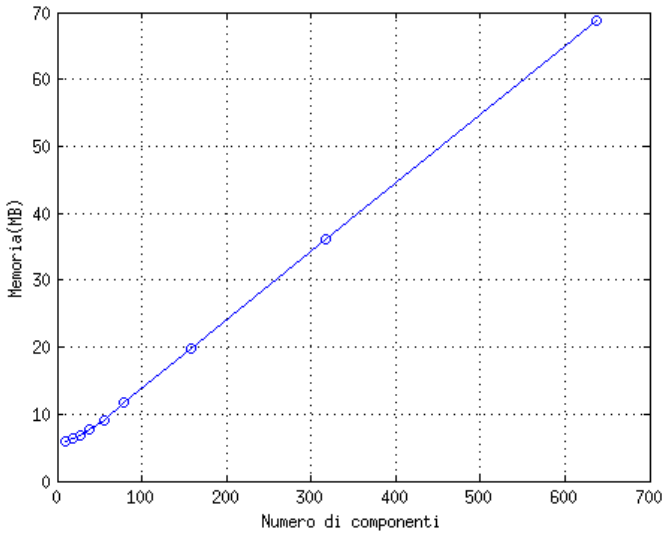
lazy, i risultati per l'istanza più grande utilizzata (638 componenti e 255 nodi) sono di 3.209s di calcolo e 68.87MB di memoria allocata. Sebbene sia chiaro che la complessità dell'algoritmo di ricostruzione del behavior sia esponenziale all'interno del singolo nodo che compone il SAC, è risultato che l'aggiunta di nodi al sistema non comporta una crescita esponenziale di risorse. Al contrario, i tempi di esecuzione si mantengono lineari nel numero di nodi del sistema e, per nodi di dimensione costante, l'andamento è lineare anche rispetto al numero di componenti. Questo andamento è facilmente osservabile nella figura 6.3(a) e 6.3(b), rispettivamente per quanto riguarda il tempo di esecuzione e la RAM utilizzata.

Componenti	Nodi	Tempo(ms)	Memoria(MB)
9	4	16.68	5.88
18	7	37.27	6.28
27	10	59.38	6.86
38	15	87.79	7.82
56	21	142	9.08
78	31	302	11.84
158	63	751	19.87
318	127	1615	36.18
638	255	3209	68.87

Tabella 6.2: Risultati dei test riguardanti il metodo lazy



(a) Tempo di esecuzione



(b) Memoria allocata

Figura 6.3: Risultati dei test riguardanti il metodo lazy

Capitolo 7

Conclusioni

Il lavoro svolto ha raggiunto gli obiettivi che erano stati prefissati. Da un lato è stato definito un nuovo insieme di sistemi a eventi discreti, chiamati sistemi attivi complessi (SAC) e ne è stata fornita una specifica formale. I SAC sono stati concepiti come estensione dei sistemi attivi tradizionali, in quanto formati da una rete di questi ultimi, secondo una topologia gerarchica. Questo modello è stato suggerito dal comportamento di molti sistemi reali che possono essere descritti a diversi livelli di astrazione. D'altro canto è stato sviluppato un software in grado di calcolare le diagnosi di questi nuovi sistemi, secondo due metodi risolutivi. Il metodo greedy è stato ottenuto come naturale ampliamento dell'algoritmo diagnostico dei sistemi attivi tradizionali, mentre il metodo lazy è stato concepito al fine di migliorarne l'efficienza. Il confronto sperimentale tra le due tecniche, infatti, ha mostrato come a fronte di una crescita delle dimensioni del sistema, l'algoritmo greedy fosse inefficiente, consumando una quantità di memoria e un tempo di esecuzione esponenziali nel numero di componenti totali. L'algoritmo lazy, invece, non possiede un comportamento siffatto, ma trae vantaggio dalla topologia del sistema e produce la diagnosi in un unico passo bottom-up lungo la gerarchia del sistema. La complessità di questo algoritmo si stabilizza alla linearità per nodi di dimensione costante, potendo risolvere problemi che attraverso il metodo greedy non sarebbero affrontabili nella pratica. A fronte di questi risultati positivi, il lavoro svolto può essere esteso in molteplici direzioni.

7.1 Sviluppi futuri

7.1.1 Parallelizzazione della diagnosi lazy

Il metodo di diagnosi lazy ricostruisce in successione i comportamenti dei singoli nodi che compongono il sistema, seguendo un ordinamento topologico dell'albero che caratterizza l'interazione dei nodi. Un ordinamento topologico totale, tuttavia, non è strettamente necessario, in quanto è sufficiente fornire un ordinamento parziale. Il calcolo dei behavior non vincolati è eseguito uno indipendentemente dall'altro; questo suggerisce in modo naturale una parallelizzazione in fase diagnostica. Una volta ricostruiti i comportamenti dei nodi foglia, allo stesso modo il livello superiore della gerarchia è composto da nodi che, sebbene siano vincolati dal comportamento dei nodi sottostanti, non sono vincolati tra loro. Anche la ricostruzione del loro comportamento, quindi, può avvenire in parallelo. Procedendo in questo modo, è possibile effettuare una diagnosi in parallelo sui diversi nodi appartenenti allo stesso livello dell'albero. Questo metodo se implementato, causerebbe un ulteriore miglioramento nelle prestazioni del calcolo, rendendo l'algoritmo linearmente dipendente dal numero di livelli nella topologia del sistema, piuttosto che dal numero di nodi totale.

7.1.2 Aumento del preprocessing

La fase di preprocessing permette di effettuare dei calcoli in fase di modellazione, evitando di spendere risorse nella successiva fase di diagnosi. L'aumento del calcolo in fase precedente potrebbe consistere nella generazione delle interfacce, la quale può richiedere delle risorse computazionali in alcuni casi costose, trattandosi di un algoritmo analogo alla subset construction. Attualmente le interfacce sono generate a partire dai behavior dei nodi, ma nulla vieta che tali interfacce possano essere generate partendo dai behavior space, che descrivono il comportamento prescindendo dall'osservazione, nota soltanto nella successiva diagnosi. Sebbene la generazione del behavior space è stata scartata nel metodo greedy poiché non praticabile nemmeno offline, la costruzione del behavior space del singolo nodo non dovrebbe essere altrettanto estenuante, assumendo una dimensione del singolo nodo limitata. In fase diagnostica le transizioni di interfaccia attuabili dovranno però essere selezionate in base all'osservazione data: parte delle interfacce costituirà una evoluzione inconsistente con l'osservazione temporale e andrà scartata. Si noti che in letteratura sono presenti metodi diagnostici che calcolano un automa, detto diagnosticatore, derivato dal behavior space dell'intero sistema. Sebbene questo renda la successiva diagnosi lineare, la generazione di tale automa offline non è praticabile per problemi di dimensione reale.

7.1.3 Variazioni nella topologia del sistema

Nell'ambito di questo lavoro di tesi è stato analizzato il caso di sistemi attivi connessi tra loro in maniera gerarchica, formando un albero. Questa topologia particolare, tuttavia, può essere facilmente estesa al caso più generale di grafo aciclico. Anche in questo contesto è possibile ricostruire il comportamento dei nodi non vincolati, proseguendo in modo analogo a quanto descritto nella trattazione. Un'attenzione particolare deve essere prestata, tuttavia, nel caso in cui un nodo abbia più di un genitore, cioè influenzi più di un altro nodo del sistema. In questo caso la ricostruzione dei behavior dei nodi genitori deve essere effettuata in maniera sincronizzata, nel senso che un cambiamento di stato dell'interfaccia relativa al nodo figlio deve valere contemporaneamente per tutti i nodi padre, altrimenti la diagnosi potrebbe avere delle soluzioni aggiuntive non sound. La ricostruzione prosegue allo stesso modo del caso di topologia ad albero: se vi sono più nodi radice, il behavior di ognuno di questi deve essere decorato, e le rispettive diagnosi combinate tra loro.
Un problema più complicato si verifica quando il sistema costituisce un generico grafo ciclico. In questo caso non è possibile effettuare semplicemente un processo bottom-up. Il procedimento consiste nella scelta di un nodo di partenza, ricostruendone il comportamento relativo e la corrispondente interfaccia; successivamente, in caso esso sia influenzato da un altro nodo, il comportamento e l'interfaccia verranno potati opportunamente, in modo da essere consistenti con i pattern event ricevuti. Il processo quindi si sviluppa percorrendo i link che collegano i nodi del sistema un numero finito di volte, fino a quando il pruning non raggiunge una situazione di stabilità.

7.1.4 Osservazioni incerte

In questo lavoro, le osservazioni dei nodi sono assunte essere sequenze lineari di label osservabili. In sistemi reali, tuttavia, un'osservazione potrebbe consistere in un ordinamento parziale delle label ricevute, anziché un ordinamento totale. In questo caso l'osservazione sarebbe percepita come un grafo orientato aciclico, invece di una semplice sequenza. Altri tipi di incertezze potrebbero riguardare il contenuto dell'osservazione: una label incerta potrebbe essere presente o meno nell'osservazione. Una volta delineato il grafo aciclico relativo all'osservazione, la consumazione di essa può essere monitorata generando un corrispondente automa, chiamato *index space*[3], nelle cui transizioni sono presenti le label dell'osservazione.

7.1.5 Monitoring

Nell'ambito di questo lavoro il focus è sulla cosiddetta diagnosi a posteriori, la quale avviene dopo un'osservazione completa del sistema, a seguito della quale viene calcolato l'insieme di diagnosi candidate. In un sistema reale come ad esempio una centrale nucleare o una rete elettrica nazionale, il sistema difficilmente può essere pensato in uno stato

quiescente a seguito del quale operare la diagnosi. L'osservazione in questi casi verrà ricevuta passo dopo passo e il compito della diagnosi (detta in questo ambito *monitoring*) consiste nel generare le diagnosi candidate in tempo reale durante l'evoluzione del sistema, a seguito di una singola label di osservazione. Estendendo la ricerca in tale direzione si potrebbe dare un metodo utile in applicazioni reali di questo tipo.

Bibliografia

[1] Aho A., Lam M., Sethi R., and Ullman J. *Compilers. Principles, Techniques and Tools*. Pearson, 2006.

[2] Cassandras C. and Lafortune S. *Introduction to Discrete Event Systems*. Springer, 2008.

[3] Lamperti G. and Zanella M. *Diagnosis of active systems. Principles and Techniques*. Kluwer Academic Publishers, 2003.

[4] Levine J. *Flex & Bison*. O'Really Media, 2009.

[5] Lamperti G. and Zhao X. Diagnosis of active systems by semantic patterns. In *IEEE Transactions on Systems, Man, and Cybernetics: systems, vol.44, no.8*, 2014.

[6] Lamperti G. and Zhao X. Diagnosis of higher-order discrete-event systems. In *A. Cuzzocrea et al.(Eds.):CD-ARES 2013, LNCS 8127, pp.162-177*, 2013.

[7] Lamperti G. Compilers. `http://gianfranco-lamperti.unibs.it/co/co.html`, 2015.

[8] Department of Computer Science University of Western Ontario(Canada). The grail+ project. `http://ftp.csd.uwo.ca/Research/grail/index.html`, 2002.

[9] Le Maout V. Astl, the automata standard template library. `http://astl.sourceforge.net/`, 2015.

[10] Dawes B., Abrahams D., and Rivera R. Boost c++ libraries. `http://www.boost.org/`, 2015.

[11] AT&T Research Labs. Graphviz - graph visualization software. `http://www.graphviz.org/`, 2015.