Note di implementazione

Translated Automata

Struttura della cartella

- Per avviare il programma tramite *Makefile*, richiamare il comando **make** all'interno della cartella "project". Il file eseguibile si trova all'interno della cartella "/bin/".
- I file contenuti nella cartella "project" contengono il codice in C++ delle varie classi. Attualmente non sono presenti classi di testing o codice di librerie esterne.
- All'interno della cartella project i file sorgenti sono contenuti all'interno della cartella "src"; i file header, invece. sono nella cartella "include".

Struttura delle classi e del progetto

Vengono descritte di seguito le principali classi che compongono il programma, accompagnate da alcune note che ne spiegano le funzionalità e le storie implementative.

- Il flusso di esecuzione è gestito all'interno del file "*main.cpp*":
  - Viene generato un automa **DFA** mediante la classe `DFAGenerator`.
  - Viene generata una **traduzione** sull'alfabeto dell'automa mediante la classe `TranslationGenerator`.
  - Viene eseguito l'algoritmo **Embedded Subset Construction** (**ESC**).
- L'implementazione degli **automi** si articola su tre diverse classi:
  - `Automaton<State>`, classe template che rappresenta un automa generico, e fornisce tutti i metodi principali.
  - Classi "`DFA`" e "`NFA`", attualmente semplici istanze template di `Automaton<StateNFA>` e `Automaton<StateDFA>` rispettivamente, senza l'aggiunta di metodi specifici.
- Si è cercato di mantenere la stessa suddivisione delle classi degli automi anche per le implementazioni dei singoli **stati** (e come si vedrà poco sotto, anche dei rispettivi generatori).
  - La classe `State<S>` rappresenta un qualunque stato di un automa a stati finiti. Mantiene al suo interno il riferimento alle transizioni entranti ed uscenti, e fornisce metodi sufficienti per la loro gestione.
    - *Nota*: l'utilizzo di una classe *template* permette un controllo compile-time della validità delle transizioni fra stati; in questo modo, infatti, uno stato di un DFA può essere collegato solo con un altro stato DFA. Questa parte di codice è fortemente influenzata dall'implementazione di Caniato.
  - `StateNFA`, classe figlia per automi non deterministici.
  - `StateDFA`, classe figlia per automi deterministici. Quando utilizzata, è richiesto al programmatore una particolare attenzione alla creazione delle transizioni, in modo tale da non avere label duplicate in uscita. A tal proposito, si veda il paragrafo sui futuri punti di sviluppo.

    - `ConstructedStateDFA`, classe figlia di `StateDFA`. Rappresenta uno stato deterministico ottenuto mediante *algoritmi di costruzione dell'automa* basati sulla trasformazione di un NFA. Viene infatti utilizzata in Subset Construction e ESC; memorizza l'**estensione** (`ExtensionDFA`) dello stato -ossia il sottoinsieme di stati NFA che tale stato rappresenta- e offre metodi ed operazioni su di essa.
- La **generazione di automi**, ancora una volta, riflette la partizione fra determinismo e non-determinismo:
    - La classe astratta template `AutomataGenerator<Automaton>` fornisce i metodi per parametrizzare la creazione di un automa e si occupa di memorizzare tali parametri.
    - Le classi figlie "`DFAGenerator`" ed "`NFAGenerator`" contengono il codice per la generazione dello specifico automa del tipo richiesto.
- Il tipo "**`Alphabet`**" non è che un alias di `vector<string>`. Si noti che:
    - Per aumentare la variabilità oltre i 26 caratteri dell'alfabeto inglese, ogni simbolo di un `Alphabet` è memorizzato come *string* e non come *char*. Questo potrebbe tuttavia generare confusione fra le stringhe nel vector (simboli dell'alfabeto) e le stringhe del linguaggio riconosciuto da un automa (parole del linguaggio, ottenute come combinazioni dei simboli dell'alfabeto).
    - Il metodo "`generateAlphabet`" di `AutomatonGenerator` permette di creare un alfabeto della cardinalità desiderata come combinazione dei caratteri prescelti.
    - **`(!)`** La stringa vuota (ossia il simbolo ε) **NON** viene considerata fra le possibili stringhe di un alfabeto, e nemmeno nel resto del programma. Pertanto gli automi NFA non presenteranno ε-transizioni.
- L'implementazione di una **traduzione** all'interno del programma è fedele alla definizione formale: è stata utilizzata una mappa che associa ad ogni simbolo dell'alfabeto di dominio un simbolo dell'alfabeto di codominio.
    - Più correttamente: la mappa mantiene in memoria solamente le associazioni differenti dall'identità, al fine di ottimizzare le dimensioni.
    - Non viene memorizzato alcun riferimento al dominio o al codominio. Una traduzione, pertanto, può operare indistintamente su qualunque stringa in ingresso, sia essa presente come chiave (e quindi verrà mappata secondo l'associazione corrispondente), sia essa appartenente a domini esterni (e quindi verrà mantenuta identica).
    - La classe `Translation` offre inoltre metodi per la traduzione di stringhe e di interi alfabeti, più metodi per la visualizzazione.
- Le classi `EmbeddedSubsetConstruction` e `SubsetConstruction` offrono un'implementazione (non ancora ottimizzata) degli algoritmi ESC e SC, più eventuali metodi per gestire operazioni interne.
- Il file "*debug.hpp*" offre alcune semplici *macro functions* per la gestione delle stampe di **debug**. Per utilizzarle è sufficiente definire la macro `DEBUG_MODE` con la direttiva al preprocessore **`#define`**:
    - All'interno del file "*debug.hpp*", per attivare le stampe di debug in tutto il progetto.

    - Prima dell'inclusione del file mediante direttiva **`#include`**, per attivarle soltanto per le funzioni definite in quel sorgente (facendo attenzione a non avere inclusioni successive).

Possibili punti da sviluppare meglio

Il seguente elenco ha il solo scopo di raccogliere alcune perplessità sorte durante l'implementazione, che *potrebbero* fungere da spunto per successivi miglioramenti o che evidenziano alcune criticità nella struttura del progetto. L'elenco non vuole assolutamente essere un tentativo di direzionare il lavoro futuro.

- La classe "`Automaton`" mantiene al suo interno i riferimenti agli stati mediante l'uso di un **set**. Questa implementazione, seppur semplice e comoda da utilizzare per evitare stati duplicati, non è particolarmente efficiente nelle operazioni di ricerca. D'altra parte l'implementazione del progetto di Caniato prevedeva una mappa <nomi, stati>, che sarebbe risultata altrettanto pesante ogni volta che si fosse reso necessario aggiornare l'estensione (procedura *Extension Update*) di uno stato DFA (=> e quindi il suo nome => e quindi la sua chiave nella mappa).
- E' possibile che alcuni metodi di alcune classi non vengano mai utilizzati all'interno del progetto; sarebbe opportuno rimuoverli, anche solo per velocizzare l'operazione di compilazione.
- L'utilizzo di classi template in C++ peggiora l'accoppiamento fra classi dal punto di vista della corretta progettazione OO. Tutte le classi genitori, infatti, sono costrette a mantenere un riferimento esplicito alle classi figlie (*Dependency Inversion, SOLID principle*) per istanziare la classe template su un particolare tipo.
- La classe `StateDFA` non aggiunge nulla di più alla classe State, ma in realtà dovrebbe sovrascrivere il comportamento del metodo "`connectChild`" in modo da permettere l'aggiunta di un solo figlio per etichetta. Questo violerebbe il *Liskov Substitution Principle*, ma sarebbe giustificato dalla comodità di implementazione. Il vero problema nasce invece quando la classe `StateDFA` viene specializzata in `ConstructedStateDFA`, la quale deve invece permettere l'aggiunta di più figli per la stessa etichetta poiché tale operazione è necessaria nella prima fase di ESC (quando l'automa deterministico D' ottenuto da *Automaton Translation* deve risultare essere isomorfo a N, che è invece un NFA). Una seconda sovrascrittura del metodo "`connectChild`" di `ConstructedStateDFA` per renderlo equivalente a quello della classe State risulterebbe completamente contraria all'idea di ereditarietà nei linguaggi OO, oltre a violare il principio di cui sopra. Per questo, l'implementazione attuale non prevede alcuna sovrascrittura in `StateDFA` e permette l'uso del metodo "`connectChild`" in maniera più libera; è compito del programmatore assicurarsi che nella generazione di un DFA (ad esempio, nella classe `DFAGenerator`) venga rispettato il vincolo sulle etichette di uno stato.
- Il punto precedente - focalizzato sul metodo "`connectChild`" - evidenzia anche come la classe `StateDFA` non aggiunga alcuna reale funzionalità al progetto, ma sia utile solamente ai fini di una migliore strutturazione del modello di dominio. L'unico uso efficace lo si riscontra in `DFAGenerator`, dove è possibile evitare l'uso di

`ConstructedStateDFA` poiché la generazione del DFA non si basa sulla trasformazione di un NFA. Valutare una riprogettazione di questo modulo.
- Potrebbe essere opportuno estrarre la generazione di un alfabeto dalla classe `AutomataGenerator` ed inserirla in una classe apposita [*extract class*]. Allo stesso tempo, potrebbe essere utile accorpare le principali funzionalità di generazione di istanze (generazione di automi, di traduzioni, di alfabeti, etc…) in un'unica classe *controller* che si occupi in toto della generazione di problemi e operante tramite delega.
- La classe `Translation` potrebbe fornire un metodo per la traduzione di interi automi che accetti in ingresso un automa di qualunque tipo e che restituisca (necessariamente) un NFA. Attualmente la traduzione di automi per l'algoritmo *ESC* è implementata all'interno di `EmbeddedSubsetConstruction`.
- Standardizzare e/o uniformare l'interfaccia di chiamata di *Tradz+SC* ed *ESC*, in modo tale da facilitare il confronto.

Bug attualmente riconosciuti

- In alcuni casi l'automa DFA generato mediante ESC contiene uno stato denominato "**}**", probabilmente con estensione vuota. Resta ancora da capire perché questo succeda.
- In una trascorsa esecuzione, l'algoritmo ESC non è giunto a terminazione e ho dovuto interromperlo manualmente. Segnalo questo fatto nonostante si tratti di un caso isolato, dovuto probabilmente a errori esterni alla classe `EmbeddedSubsetConstructor` (forse `DFAGenerator`, su cui stavo lavorando in quel momento?).