UNIVERSITÀ DEGLI STUDI DI PISA

Dipartimento di Filologia, Letteratura Linguistica

CDL Informatica Umanistica

Laurea Magistrale

UNIVERSITÀ DI PISA

UN DOMAIN SPECIFIC LANGUAGE PER LA MODELLAZIONE E L'ANALISI DI GIOCHI

RELATORE Prof. Paolo MILAZZO

RELATORE Dott. Giovanni PARDINI

CONTRORELATORE Prof.ssa Maria Eugenia OCCHIUTO

Candidato Giovanna BROCCIA

ANNO ACCADEMICO 2014-15

Riassunto

Le tecniche di modellazione e di model checking permettono di modellare sistemi e di verificarne proprietà attese o desiderate.

In un precedente lavoro di sperimentazione, tali tecniche sono state utilizzate per modellare dei giochi da tavolo; per farlo è stato utilizzato il software PRISM.

Analizzando i modelli di gioco, tuttavia, ci si è resi conto che il linguaggio di input di tale software poco si adatta alla descrizioni di alcune caratteristiche fondamentali dei giochi da tavolo, essendo molto semplice.

È stato quindi ideato e creato un Domain Specific Language, chiamato GameDSL, per rendere la modellazione di giochi più semplice e concisa.

Nell'introduzione di questo elaborato si intende inquadrare il lavoro svolto e fornire uno stato dell'arte in tale settore.

Nel secondo capitolo si intende dare un quadro introduttivo generale ai linguaggi formali, partendo dai concetti base fino ad arrivare a concetti più complessi.

Nel terzo e nel quarto capitolo verranno presentate le tecniche di model checking e lo strumento PRISM, nonché i modelli di gioco implementati con tale strumento.

Il quinto capitolo presenta i vantaggi e gli svantaggi della modellazione tramite PRISM e i vantaggi dell'implementazione di un Domain Specific Language per la modellazione di giochi, prendendo in esempio i modelli creati con PRISM e sollevando delle problematiche che, con un linguaggio di più alto livello, si potrebbero risolvere facilmente.

Nel sesto e settimo capitolo viene presentato GameDSL, con la sua grammatica e con i modelli di gioco re-implementati.

Infine nell'ottavo capitolo si presenta l'implementazione del parser tramite lo strumento JavaCC e i risultati ottenuti dalla verifica sui modelli creati con GameDSL.

Indice

1 INTRODUZIONE

1.1 Motivazioni

Il model checking [1] consiste in tecniche di verifica formale che consentono di controllare il comportamento di un dato sistema, sulla base di un modello del sistema in questione, tramite ispezioni sistematiche a tutti i suoi stati e verifiche formali alle proprietà attese o desiderate.

Ipotizzando di avere un sistema che può avere un insieme finito di stati diversi, il modello che lo descrive sarà un sistema di transizioni in cui:

- gli elementi che costituiscono il sistema sono descritti da un insieme di variabili;
- le combinazioni dei possibili valori delle variabili descrivono l'insieme degli stati in cui può trovarsi il sistema;
- coppie di stati possono essere messe collegate da una relazione di transizione per indicare la possibilità per il sistema di passare da uno stato all'altro.

Consideriamo un semplice esempio: il sistema considerato è un forno a microonde che può avere quattro possibili configurazioni, o stati, e per il quale sono fornite le seguenti specifiche:

- Il forno si trova inizialmente in uno stato in cui non ha ricevuto nessun comando e non sta compiendo nessuna azione, la sua porta è aperta;
- Il forno riceve il comando start ma non può eseguire nessuna azione perché la porta non è stata chiusa;
- Il forno riceve il comando start e la porta è chiusa: si avvia la cottura durante la quale il forno non può ricevere altri comandi. A fine cottura la porta si apre automaticamente e il forno ritorna allo stato iniziale.

Gli elementi del forno descritti dalle variabili sono *close*, *start* e *cooking,* a seconda del valore di verità che assumono vengono definiti quattro diversi stati:

1. S_0 = *!close, !start, !cooking*

   Il forno è nello stato iniziale: la porta è aperta e non ha ricevuto nessun comando.

2. S_1 = *!close, start, !cooking*

   Il forno ha ricevuto il comando start ma la porta non è chiusa dunque non cucina.

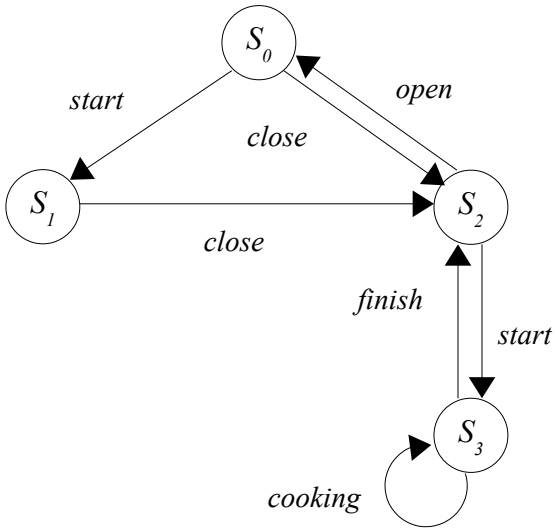
3. S_2 = *close, !start, !cooking*

   Il forno è chiuso ma non ha ricevuto nessun comando, ergo non sta cucinando.

4. S_3 = *close, start, cooking*

   Il forno è chiuso, ha ricevuto il comando start e sta cucinando.

Di seguito il modello che rappresenta le specifiche suddette.



Esaminando tutti gli stati raggiungibili da questo sistema è facile convincersi che non è possibile che il forno sia in cottura mentre la porta è aperta.

Nelle ultime due decadi queste tecniche si sono sviluppate notevolmente e sono state utilizzate, quasi esclusivamente, nel contesto della *Computer Science*, soprattutto nell'ambito della verifica dei circuiti elettronici, ad esempio dei processori, e della

verifica di programmi (*software model checking*), nel quale viene usato come metodo per trovare bachi.

Non è, tuttavia, difficile capire come questo genere di verifiche siano congeniali a diversi tipi di prodotti e che la costruzione di modelli ad hoc e la verifica formale di proprietà desiderate, sia una tecnica utile in diversi campi del sapere.

Il lavoro di sperimentazione alla base di tale tesi nasce da un precedente lavoro di sperimentazione [2] [3], in cui il model checking probabilistico è stato utilizzato nell'ambito del *game design*, disciplina che mira al perfezionamento delle metodologie di progettazione di artefatti ludici. Al fine di aiutare il progettista ad aumentare una delle principali caratteristiche che i giochi dovrebbero avere secondo i *game studies,* la *replayability*, sono stati modellati tra giochi da tavolo e sono state verificate determinate proprietà. La *replayability*, letteralmente, non è altro che la "rigiocabilità" di un gioco, ovvero la caratteristica di un gioco che offre un'esperienza significativa al punto che, chi ci ha giocato, deciderà di rifarlo. Tale caratteristica è influenzata da altre cinque caratteristiche: la durata del gioco, la varietà del gioco, la randomizzazione delle partite e delle sue fasi, le differenti strategie vincenti, l'abilità di gioco cooperative e competitive.

Ognuno dei tre giochi studiati rappresenta un diverso tipo di gioco, secondo classificazioni indagate nei *game studies*. Il primo è il Gioco dell'Oca, gioco da tavolo caratterizzato da dinamiche fortemente aleatorie; il secondo è un gioco di strategia ispirato al Risiko (d'ora in avanti ci riferiremo ad esso con il nome Risiko), caratterizzato da una componente strategica e una puramente aleatoria; il terzo è un gioco del tipo dungeon crawl (d'ora in avanti ci riferiremo ad esso con il nome Dungeon Crawl), gioco collaborativo in cui l'esperienza di gioco non è legata al singolo giocatore ma è un'esperienza di gruppo. Per quel che riguarda le proprietà verificate, nel primo modello si è indagato su quale sia il numero di caselle penalizzanti tali da stabilizzare o meno le dinamiche di gioco; nel secondo modello si è indagato su possibili approcci alla partita (neutro, aggressivo e passivo) e si è inteso osservare il variare della durata del gioco e il variare delle probabilità di vittoria al variare di tali approcci; nel terzo modello si è indagato su alcune proprietà che caratterizzano un gioco collaborativo e si è affrontato uno dei maggiori problemi riscontrabile in giochi di tale genere, ovvero l'eliminazione precoce di uno dei giocatori.

Per modellare tali casi di studio è stato usato il software PRISM model checker, uno strumento che permette di creare modelli come *Markov chains* e di descrivere proprietà in una logica temporale.

Il linguaggio di PRISM si è, però, dimostrato inadeguato per descrivere alcuni aspetti comuni ai giochi, come le strategie dei giocatori, l'ambiente di gioco e le scelte probabilistiche con distribuzioni legate allo stato del gioco. Il linguaggio è molto semplice: si possono definire delle variabili e delle espressioni ma non è possibile rappresentare una struttura dati più complessa o avere dei comandi che permettono di iterare una determinata azione. Andando a valutare i modelli svolti – allegati in Appendice – si nota subito come la mole di codice scritto sia molto ampia anche laddove si vuole gestire eventi del gioco relativamente semplici.

Da questa riflessione nasce l'idea da cui parte il lavoro di sperimentazione svolto: definire un nuovo linguaggio di alto livello, più ricco rispetto a quello di PRISM, che renda più semplice e rapida la costruzione dei modelli.

A tale scopo è stato scelto di definire un Domain Specific Language, ovvero un linguaggio di programmazione o di specifica dedicato a particolari problemi di dominio, a una particolare tecnica di rappresentazione e/o una particolare soluzione tecnica; in sostanza un linguaggio che si occupa di un dominio specifico, nel nostro caso la modellazione di giochi. Tale linguaggio è stato chiamato GameDSL (Game Domain Specific Language, appunto).

1.2 Stato dell'arte

Il software appena descritto, PRISM, non è ovviamente l'unico strumento in grado di fare tale tipo di analisi, tuttavia si è rivelato più congeniale definire un linguaggio specifico per questo genere di modellazioni, a fronte delle problematiche presenti e con il linguaggio di PRISM e con gli altri model checker disponibili. Questi strumenti presentano generalmente due tipi di problematiche: o non sono model checker probabilistici, oppure hanno un linguaggio non di alto livello paragonabile a quello di PRISM, se non anche inferiore in alcuni casi.

Uno degli esempi più calzanti in questo senso è rappresentato da un'estensione di PRISM, chiamata PRISM-games [4]. Questo strumento è stato creato per la verifica di sistemi probabilistici che possono incorporare comportamenti competitivi o collaborativi, modellati come giochi stocastici multi-giocatore (*Stochastic Multipalyer Games*, SMGs). Tale tipo di modello potrebbe essere visto come una generalizzazione dei processi decisionali di Markov (*Markov Decision Process*, MDPs), nei quali le scelte non deterministiche sono sotto il controllo di diversi giocatori. Appare dunque

chiaro che tale estensione sarebbe risultata più congeniale nello sviluppo di modelli di giochi se non fosse che il linguaggio di PRISM-games risulta ancora di basso livello rispetto a un linguaggio che ci permetta di modellare i giochi e le loro caratteristiche con più facilità.

Di seguito verranno descritti i principali model checker esaminati, con una breve spiegazione sul perché non sono congeniali alla modellazione e l'analisi di artefatti ludici.

1. Model checker non probabilistici:

    - Uno dei model checker più utilizzati è Spin [5], un programma di verifica usato principalmente per fare software model checking, ovvero modellazione e analisi di programmi al fine di trovare bachi. Questo strumento è stato sviluppato dal gruppo Unix e presenta un linguaggio di input, chiamato PROMELA che, anche a un primo sguardo, appare un linguaggio di alto livello. Tuttavia il model checker Spin non è uno strumento in grado di fare verifiche probabilistiche. Tale caratteristica risulta essere limitante nello studiare i casi proposti, dal momento che le verifiche formali fatte su di essi, erano di tipo probabilistico. Di fatto, Spin risulta essere un ottimo strumento per debuggare dei programmi software ma non è adatto all'ambito in questo lavoro di sperimentazione.

    - Un altro strumento di modellazione è NuSMV [6], un model checker sviluppato in ambito accademico presso l'Università di Trento dalla re-implementazione e estensione di SMV – model checker sviluppato alla Carnegie Mellon University (Università privata di Pittsburgh, in Pennsylvania). NuSMV è basato su diagrammi di decisione binaria (BDDs). Lo strumento possiede un linguaggio molto simile a quello di PRISM, dunque non di alto livello; inoltre anch'esso, come il procedente, non è un model checker probabilistico.

2. Model checker con linguaggi base [7]:

I model checker presentati in questa sezione provengono tutti da ambiti accademici: sono tutti strumenti scaricabili gratuitamente, non nati al fine della vendita. Questo aspetto, unito anche all'utilizzo più di nicchia che di essi viene fatto, rende chiaro il motivo per cui il loro linguaggio non mira alla semplificazione per il cliente/utente e rimane un linguaggio di base, in alcuni casi difficile da capire ed usare.

- ETMCC sviluppato dal gruppo "*Stochastic Modelling and Verification*" dell'Università di Erlangen-N¨urnberg, in Germania, e dal gruppo "*Formal Methods & Tools*" dell'Università di Twente, in Olanda. Tale strumento è un model checker che svolge verifiche su modelli descritti esclusivamente come catene di Markov a tempo continuo (*Continuous-Time Markov Chain*, CTMC). Lo strumento offre un'interfaccia grafica sia per la modellazione di sistemi che per la scrittura di proprietà da verificare.

- MRMC sviluppato dal gruppo "*Software Modelling and Verification*" all'Università RWTH di Aachen, in Germania, e dal gruppo "F*ormal Methods & Tools*" dell'Università di Twente, in Olanda. Tale strumento è molto simile a PRISM, difatti consente di descrivere modelli come catene di markov a tempo continuo (*Continuous-Time Markov Chain*, CTMC) e discreto (*Discrete-Time Markov Chain*, DTMC) e come estensioni di queste ultime ma con i *rewards*; inoltre la logica temporale utilizzata per descrivere le proprietà è la stessa usata in PRISM, la *Probabilistic Computation Tree Logic* (PCTL).

- YMER sviluppato all'Università Carnegie Mellon di Pittsburgh in Pennsylvania. Lo strumento supporta la descrizione di modelli come catene di markov a tempo continuo (*Continuous-Time Markov Chain*, CTMC) e come *Generalized Semi-Markov Processes* (Processi di semi-Markov generalizzato GSMPs). YMER sviluppa tecniche di model checking statistico, basato su simulazioni di eventi, tuttavia può solo fornire garanzie probabilistiche di correttezza, in sostanza verifica semplicemente che il modello non arrivi ad uno stato di deadlock. Tale strumento è disponibile solo da riga di comando; inoltre il suo linguaggio è un sottoinsieme del linguaggio di PRISM.

- VESTA sviluppato all'Università dell'Illinois. Come il precedente è un model checker probabilistico in grado di fornire garanzie probabilistiche di correttezza. Anche tale strumento descrive modelli come CTMC e DTMC. Il suo linguaggio è basato su Java, mentre le proprietà sono espresse anche in questo caso in PCTL.

Da tali presupposti è evidente che l'utilizzare un altro strumento di model checking non avrebbe comunque risolto le problematiche incontrate con il linguaggio di PRISM, e appare dunque chiaro che, l'implementazione di GameDSL sia la soluzione ottimale per

la rimodellazione di tali casi studio.

Inoltre, ipotizzando una futura implementazione di un compilatore che si occupi della traduzione dal suddetto Domain Specific Language al linguaggio di input di PRISM, si potrebbero semplificare le verifiche avendo, da un lato, dei modelli più concisi, semplici e adatti alla descrizione di giochi (definiti tramite GameDSL), e dall'altro la praticità e l'efficienza delle verifiche probabilistiche svolte con PRISM.

Il model checking nel campo del game design appare come un ambito di ricerca pressoché inesplorato, soprattutto se si parla di meccaniche di gioco.

Allo stato dell'arte alcuni studi sono stati compiuti sul model checking nell'ambito dei giochi, ma non sono molto numerosi e la maggior parte di essi si occupa di modellazione di videogiochi per esplorare altri tipi di caratteristiche: il focus di tali studi è su aspetti più specifici della progettazione e dell'implementazione – per assicurare il non verificarsi di situazioni di deadlock, ad esempio; in sostanza su aspetti più legati al software che alle dinamiche di gioco.

Lo studio svolto in [8], ad esempio, si occupa di model checking di video games di avventura, per assicurarsi in fase di design che il gioco non vada in stallo. Tale studio è stato svolto con i videogiochi di e-Adventure, una piattaforma per lo sviluppo di videogiochi di avventura educativi, descritti usando un Domain Specific Language orientato alla scrittura di giochi. Tale linguaggio viene tradotto in specifici modelli che vengono poi verificati automaticamente tramite il model checker NuSMV, presentato poco sopra, per accertarsi che non si verifichino stati di stallo durante il gioco. Tale approccio facilita le verifiche che altrimenti verrebbero fatte a mano e richiederebbero molto tempo.

Sempre in tale contesto si inserisce lo studio svolto in [9]. In questo caso i videogiochi sono scritti in Java e viene utilizzato il model checker Java Pathfinder (JPF), uno strumento atto a verificare programmi eseguibili Java. Lo studio in sostanza verifica la presenza di bachi nell'implementazione del videogiochi, utilizzando due estensioni di Java Pathfinder che sono jpf-probabilistic e jpf-nhandler.

Gli altri tipi di analisi svolte nel campo dei *game studies* non riguardano le verifiche di giochi ma più che altro la creazione. La maggior parte dei lavori di sperimentazione ha a che fare con la creazione di motori di gioco.

Un esempio di tali tipi lavori è quello sviluppato presso il dipartimento di C*omputer*

Science dell'Università di Alberta, in Canada [10].

Nel suddetto lavoro è stato creato un Domain Specific Language, chiamato PhyDSL-2, per la creazione di un motore di gioco che sia in grado di facilitare la progettazione di artefatti ludici basati sulla fisica bidimensionale. Il motore di gioco consente agli sviluppatori di riutilizzare le risorse da giochi precedentemente sviluppati, quindi facilitando le sfide dell'ingegneria del software nello sviluppo di giochi, e rendendo l'implementazione meno costosa e tecnologicamente più efficiente.

PhyDSL-2 è, in sostanza, un linguaggio di sviluppo per videogiochi pensato per persone non esperte nella programmazione, che si basa su cinque domande a cui, chi costruisce l'artefatto ludico, dovrebbe rispondere: (i) chi è il giocatore?, (ii) quali azioni sono disponibili per il giocatore?, (iii) qual è l'ambiente di gioco in cui vive il giocatore?, (iv) quali sono gli obbiettivi del giocatore?, (v) quali sono le sfide del giocatore?. Rispondendo a queste cinque domande lo sviluppatore è in grado di definire tutti gli aspetti fondamentali del gioco.

Un altro esempio in questa direzione è il lavoro presentato in [11]; in questo caso però il motore di gioco creato, chiamato LUDACORE, si basa sulla logica formale usata nell'ambito dell'intelligenza artificiale (*Artificial Intelligence*, A.I.). Tale strumento è in grado di generare tracciati di gioco che illustrano il comportamento del giocatore; in sostanza è in grado di generare artefatti ludici tramite una programmazione logica.

2 BACKGROUND SUI LINGUAGGI FORMALI

Per linguaggio formale, in matematica, logica, informatica e linguistica, si intende un insieme di stringhe di lunghezza finita formate attraverso un alfabeto finito, ovvero un'insieme finito di frasi generate attraverso un insieme finito di oggetti che vengono chiamati caratteri, simboli o lettere.

Un linguaggio formale può essere definito in diversi modi:

- attraverso una grammatica G;
- attraverso un automa;
- attraverso delle espressioni regolari.

Il seguente capitolo intende fornire una presentazione di tali metodi e si intendono chiarire alcuni dei concetti base utili alla comprensione del lavoro svolto. Per ulteriori delucidazioni si rimanda a [12], [13], [14], [15].

2.1 Concetti di base

2.1.1 Insiemi

Come già detto, un linguaggio formale è un insieme finito di simboli. Definiamo e chiariamo che cosa è un insieme.

Un insieme è una collezione di oggetti distinguibili, chiamati membri o elementi, ad esempio, possiamo avere un insieme *A* di lettere, un insieme *B* di numeri, un insieme *C* di stringhe:

$$A = \{a, b, c, d\}$$

$$B = \{1, 7, 5, 9\}$$

$$C = \{cane, gatto, topo\}$$

Se un elemento a è un elemento di un insieme A, si scrive $a \in A$ e si legge "*a appartiene ad A*"; se b non appartiene ad A, invece, si scrive $b \notin A$.

Un insieme può essere descritto in diversi modi:

- Rappresentazione estensionale (in extenso), ovvero tramite l'elenco dei suoi elementi tra parentesi graffe, utile per definire insieme finiti:

$$A = \{1, 3, 6, 8\}$$

- Rappresentazione intensionale (in intenso), ovvero identificando l'insieme costituito dagli elementi x che rendono vera una data espressione E, la quale dipende da x e ha valori booleani:

$$A = \{n \in N \mid n \% 2 = 0 \}$$

- Rappresentazione induttiva, ovvero tramite uno o più casi base, detti assiomi, e una o più regole di inferenza definite nella forma "se-allora":

assioma:

$$0 \in Pari$$

regole di inferenza:

$$\frac{n \in Pari}{n+2 \in Pari}$$

Due insiemi sono uguali se contengono gli stessi elementi: in questo caso si usa l'operatore =, ad esempio $\{3, 1, 2\} = \{1, 2, 3\} = \{3, 2, 1\}$, o anche $A=B$.

Alcuni insiemi hanno un nome particolare che li identifica univocamente: ad esempio $\mathbb{N}$ rappresenta l'insieme dei numeri naturali, $\mathbb{Z}$ rappresenta l'insieme dei numeri interi e $\mathbb{R}$ rappresenta l'insieme dei numeri reali.

Se tutti gli elementi di A sono contenuti nell'insieme B, allora si dice che A è un sottoinsieme di B e si scrive $A \subseteq B$; se $A \subseteq B$ ma $A \neq B$ allora si dice che l'insieme A è un sottoinsieme proprio di B e si scrive $A \subset B$.

Dati due insiemi A e B, si possono definire nuovi insiemi applicando le seguenti operazioni:

- Intersezione: l'insieme di tutti gli elementi appartenenti sia ad A che a B

  $A \cap B = \{x \mid x \in A, x \in B\}$

- Unione: l'insieme di tutti gli elementi appartenenti ad A oppure a B

  $A \cup B = \{x \mid x \in A \text{ oppure } x \in B\}$

- Differenza: l'insieme di tutti gli elementi appartenenti ad A che non sono appartenenti a B

  $A-B = \{x \mid x \in A, x \notin B\}$

Spesso gli insiemi che si considerano sono sottoinsiemi di un insieme U più grande, detto Universo. Dato un universo U, si definisce il complemento di un insieme A come $\bar{A} = U - A$.

Il prodotto cartesiano di due insiemi A e B, denotato da $A \times B$, è l'insieme di tutte le possibili coppie ordinate tali che il primo elemento della coppia è un elemento di A ed il secondo elemento è un elemento di B, più formalmente:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

2.1.2 Relazioni

Una relazione R su due insiemi A e B è un sottoinsieme del prodotto cartesiano $A \times B$, ovvero un insieme di coppie in cui il primo elemento viene da A e il secondo elemento viene da B. Ad esempio se:

$$A=\{a, b , c, d\}$$

$$B=\{1, 3, 2, 9, 7\}$$

allora:

$$R \subseteq A \times B$$

$$R = \{(a, 1), (a, 3), (b, 2), (c, 9)\}$$

Un esempio di relazione è la relazione “minore di” sui numeri naturali:

$$< \subseteq N \times N$$

ovvero il sottoinsieme di coppie in cui sia il primo elemento che il secondo appartengono ai numeri naturali e in cui il primo elemento è minore del secondo; scritto con la rappresentazione intensionale:

$$< = \{(a, b) \mid a, b, c \in N\ ,\ c \neq 0,\ a+c = b\ \}$$

e scritto nella notazione infissa:

$$a < b$$

2.1.3 Funzioni

Dati due insiemi A e B, una funzione f è una relazione binaria su $A \times B$, tale che per tutti gli $a \in A$, esiste soltanto un solo $b \in B$, tale che $(a, b) \in f$. L'insieme A è chiamato il dominio di f e l'insieme B è chiamato il codominio di f.

Intuitivamente la funzione f assegna un elemento di B ad ogni elemento di A; nessun elemento di A è assegnato a due distinti elementi di B, ma lo stesso elemento di B può essere assegnato a due distinti elementi di A.

Data una funzione $f(a) = b$ si dice che a è l'argomento di f e b è il valore di f per a.

Un'esempio di funzione è la funzione “successore di”:

$$succ \subseteq N \times N$$

è un sottoinsieme del prodotto cartesiano tra gli insiemi dei numeri naturali e solitamente viene scritta in notazione prefissa:

$$succ(0) = 1$$

2.2 La sintassi del linguaggio

I linguaggi di programmazione, così come tutti i linguaggi usati per la descrizione

formale di fenomeni di qualunque tipo, coinvolgono due aspetti distinti, che è importante distinguere: la sintassi e la semantica.

La sintassi ha a che fare con la struttura (o la forma) dei programmi esprimibili in un dato linguaggio. La semantica, invece, ha a che fare con il significato dei programmi esprimibili in un dato linguaggio. Se si considera il linguaggio naturale – l'italiano per esempio – due frasi come “il bambino mangia la mela” e “la mela mangia il bambino” sono entrambe sintatticamente corrette in quanto sono entrambe costruite secondo lo schema soggetto-verbo-complemento oggetto, ma solo la prima è semanticamente corretta, ovvero ha un significato ragionevole.

I linguaggi naturali posseggono un insieme di elementi di base che sono le parole e che vengono usate per formare delle frasi. I linguaggi formali, invece, posseggono un alfabeto di elementi base, detti anche elementi terminali, che sono caratteri, che si combinano insieme formando parole, anche riferite con il termine stringhe. Dunque il linguaggio formale è un sottoinsieme di tutte le possibili stringhe ottenibili combinando gli elementi terminali che rispettano determinate regole sintattiche.

Supponendo di avere un alfabeto $\Sigma=\{a, b, c\}$, il linguaggio potrà essere $L = \{abc, ccaabb, bac, cb....\}$, ovvero un insieme arbitrario di parole costruite combinando, anche più volte, i simboli del mio alfabeto, ottenendo un insieme finito o infinito.

Questo non avviene solamente per i linguaggi naturali, basti pensare al linguaggio dell'aritmetica: se si considera l'alfabeto $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, \times, /\}$, ci sono stringhe corrette sintatticamente ($4 + 5$) e altre che, nonostante siano formate a partire dall'alfabeto del linguaggio, non sono ammissibili ($3 / \times 6$) .

Stando a quanto detto, descrivere un linguaggio significa descrivere le stringhe del linguaggio stesso, ovvero avere un metodo per decidere quali stringhe fanno o meno parte del linguaggio, oppure costruire direttamente l'insieme, enumerando le stringhe che ne fanno parte. In altre parole è necessario identificare le stringhe ammissibili che compongono un linguaggio.

Per fare ciò, esistono, come già detto, tre approcci:

- il primo approccio si basa su uno strumento detto grammatica, in grado di generare tutte e sole le stringhe che fanno parte del linguaggio;

- il secondo approccio si basa su uno strumento detto automa, in grado di accettare

tutte e sole le stringhe del linguaggio;

- il terzo approccio si basa su degli strumenti detti espressioni regolari che rappresentano, usando particolari operatori, tutte e sole le stringhe che fanno parte del linguaggio.

2.3 La grammatica

Una grammatica formale G è una quadrupla $G = \{N, \Sigma, S, P\}$, dove:

- N è un insieme finito di simboli che rappresentano categorie sintattiche o simboli non terminali, generalmente rappresentati da lettere maiuscole;

- Σ è un insieme finito di simboli terminali, ovvero l'alfabeto, generalmente rappresentati da lettere minuscole;

- S è un simbolo di categoria sintattica indicato come iniziale o principale, detto assioma e appartenente a N;

- P è un insieme di regole di produzione con un lato sinistro e un lato destro. Il lato sinistro della regola di produzione è detto testa, il lato destro è detto corpo. Una regola di produzione può essere applicata ad una parola, rimpiazzando il lato sinistro con quello destro. Il corpo di una produzione può anche essere vuoto, in tal caso la produzione viene scritta $S \rightarrow \varepsilon$. Una derivazione è una sequenza di regole di produzione.

Il compito di una grammatica è quello di generare stringhe di simboli terminali. Il processo di generazione delle stringhe può avvenire tramite derivazioni o tramite la costruzione di alberi di derivazione.

Definiamo il concetto di derivazione di una stringhe tramite le regole di produzione. Ipotizziamo di avere un alfabeto $\Sigma = \{a, b\}$, l'insieme dei simboli non terminali $N = \{S\}$ e le seguenti regole di produzione:

$S \rightarrow aSb$

$S \rightarrow ab$

La derivazione avviene in questo modo:

1. Partendo dal simbolo iniziale S viene utilizzata la prima regola di derivazione:

$$S \rightarrow aSb$$

2. Viene utilizzata nuovamente la prima regola di derivazione sulla stringa ottenuta:

$$S \rightarrow aSb \rightarrow aaSbb$$

3. Viene utilizzata la seconda regola di derivazione e si ottiene la stringa (parola) che appartiene al linguaggio $L(G)$:

$$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaabbb$$

la stringa ottenuta $aaabbb \in L(G)$.

Il secondo modo per generare una stringa tramite la grammatica è quello di costruire un albero di derivazione. Supponendo di avere una grammatica G, un albero di derivazione di G è un albero in cui i nodi sono etichettati con i simboli della grammatica $\Sigma \cup N$ (terminali e non terminali) e sono rispettate le seguenti regole:

- La radice dell'albero è etichettata con il simbolo iniziale (S)
- Ogni foglia dell'albero è etichettata con un simbolo terminale
- Ogni nodo interno dell'albero è etichettato con un simbolo non terminale A i cui figli, presi da sinistra a destra, sono etichettati con i simboli della parte destra di una qualche produzione presente in P

La stringa che si ottiene concatenando i simboli associati alle foglie di un albero di derivazione, andando da sinistra a destra, si chiama stringa associata all'albero di derivazione e appartiene al linguaggio $L(G)$, ovvero il linguaggio generato da G.

Ipotizziamo di avere un alfabeto $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), +, -, \times\}$ e l'insieme dei simboli non terminali $N = \{E\}$, ovvero formato dal solo simbolo iniziale E. Le regole di produzioni sono le seguenti:

$E \rightarrow E + E$

$E \rightarrow E \times E$

$E \rightarrow (E)$

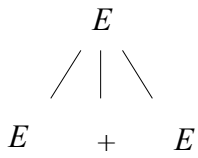
$E \rightarrow -E$

$E \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

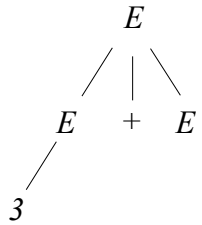
La creazione dell'albero di derivazione avviene in questo modo:

1. Il simbolo iniziale *E* è posto come radice dell'albero.

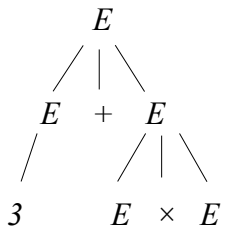
2. Utilizzando la prima regola di produzione vengono generati tre figli:



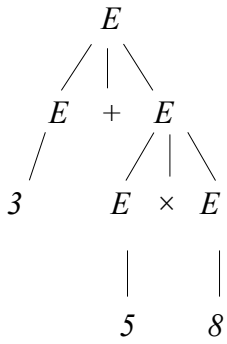
3. Sul primo figlio viene utilizzata quinta regola di derivazione e viene generato uno dei simboli terminali:



4. Sul terzo figlio viene utilizzata la seconda regola di produzione:



5. Sui figli generati viene utilizzata la quinta regola di produzione:



Leggendo da sinistra a destra i figli dell'albero, possiamo trovare la stringa *3 + 5 × 8* appartenente al linguaggio *L(G)*.

2.3.1 Classificazione di Chomsky

Le grammatiche formali sono spesso denominate Grammatiche di Chomsky, perché furono introdotte dal linguista Noam Chomsky con lo scopo di rappresentare i procedimenti sintattici elementari che sono alla base della costruzione di frasi della lingua inglese. Pur essendosi presto rivelate uno strumento inadeguato per lo studio del linguaggio naturale, le grammatiche formali hanno un ruolo fondamentale nello studio delle proprietà sintattiche dei programmi e dei linguaggi di programmazione.

Tra il 1956 e il 1959, nelle due opere “Three models for description of language, “I.R.E. Transaction on Information Theory” (1956) e “On certain formal properties of

grammars, Information and control" (1959), il linguista formulò una vera e propria gerarchia di grammatiche formali, anche dette grammatiche a struttura sintagmatica (phrase structure grammars). Chomsky fu, dunque, il primo a formalizzare le grammatiche generative – ovvero un insieme di regole che generano in modo ricorsivo le formule ben formate (fbf, ovvero una stringa che rappresenti un'espressione sintatticamente corretta e che viene definita mediante le regole della grammatica del sistema formale stesso) di un linguaggio – e fu anche il primo a classificare queste grammatiche in quattro tipi.

Secondo la gerarchia si individuano i seguenti livelli:

1. Grammatiche illimitate (di tipo-0);

2. Grammatiche context-dependent (di tipo-1);

3. Grammatiche context-free (di tipo-2);

4. Grammatiche regolari (di tipo-3).

Le grammatiche illimitate, definiscono la classe di linguaggi più ampia possibile[1]. In esse le produzioni possono avere qualunque forma, ad esempio:

$$aAb \rightarrow cdB$$

dove A e B sono simboli non terminali e $abcd$ sono stringhe di simboli terminali e non. Queste grammatiche ammettono anche la produzione di stringhe vuote e sono anche detti linguaggi di tipo-0.

Ipotizzando di avere una grammatica G in cui:

$$\Sigma = \{a, b\}$$

$$N = \{S, A\}$$

le cui regole di produzione P sono:

$$S \rightarrow aAb$$

1 Più ampia nell'ambito dei linguaggi descrivibili con grammatiche. Esistono tuttavia linguaggi per cui non esiste una grammatica corrispondente.

$$aA \rightarrow aaAb$$

$$A \rightarrow \varepsilon$$

genera il linguaggio $L = \{a^n b^n \mid n \geq 1\}$.

Le grammatiche context-dependent, anche dette context-sensitive, generano linguaggio dipendenti dal contesto e ammettono regole nella forma $\alpha S\beta \rightarrow \alpha\gamma\beta$, dove α, γ e β sono stringhe di simboli terminali e non, le stringhe α e β possono essere anche vuote e S è un simbolo non terminale. Il termine "linguaggio contestuale", deriva dal fatto che, storicamente, questi linguaggi sono stati definiti da Chomsky come la classe dei linguaggi generabili da grammatiche aventi produzioni "contestuali" del tipo $\alpha S\beta \rightarrow \alpha\gamma\beta$ in cui si esprime il fatto che la produzione $S \rightarrow \gamma$ può essere applicata solo se S si trova nel contesto $\alpha S\beta$.

Le grammatiche context-free generano linguaggi liberi dal contesto. Ammettono produzioni del tipo $S \rightarrow \gamma$, dove S è un simbolo non terminale e γ è una stringa di simboli terminali e non; cioè produzioni in cui ogni non terminale S può essere riscritto in una stringa γ indipendentemente dal contesto in cui esso si trova.

Le grammatiche regolari, anche dette grammatiche lineari, generano linguaggi regolari. Il termine lineare deriva dal fatto che al lato destro di ogni produzione compare al più un simbolo non terminale che segue o precede un simbolo terminale, ma mai entrambe le cose, e al lato sinistro della regola al più un simbolo non terminale. È possibile generare dal simbolo non terminale una stringa vuota (ε). Le regole in sostanza hanno la seguente forma: $S \rightarrow aT$. Questi linguaggi sono esattamente tutti i linguaggi riconosciuti da un automa a stati finiti. Inoltre questa famiglia di linguaggi formali può essere ottenuta con espressioni regolari.

Ipotizzando di avere una grammatica G, in cui:

$$\Sigma = \{a, b\}$$

$$N = \{S, T\}$$

le cui regole di produzione P sono:

$$S \rightarrow \varepsilon \mid aS \mid T$$

$$T \rightarrow bT \mid \varepsilon$$

Tale grammatica genera un linguaggio regolare $L = \{a^*b^*\}$.

Si verifica che per ogni $0 \leq n \leq 2$, ogni grammatica di tipo $n+1$ è anche di tipo n, pertanto l'insieme dei linguaggi di tipo n contiene tutti i linguaggi di tipo $n+1$, formando quindi una gerarchia, appunto la Gerarchia di Chomsky. È possibile mostrare che il contenimento è stretto, come mostrato nell'Illustrazione 2.1.

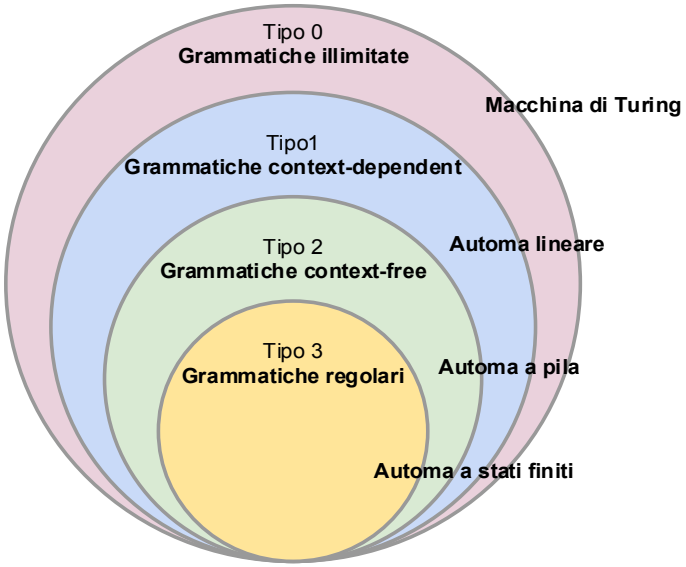


Illustrazione 2.1: Gerarchia di Chomsky

2.3.2 Notazione Backus-Naur Form

La Backus-Naur Form (BNF) è una metasintassi attraverso la quale è possibile descrivere le regole delle grammatiche *context-free* (o anche libere dal contesto).

Le regole di derivazione nella Backus-Naur Form sono espresse nei seguenti modi:

$$S \rightarrow aSb$$

$$S ::= ab$$

I simboli → e ::= sono detti metasimboli alternativi che ricoprono un ruolo speciale, ovvero servono per esprimere la regola di derivazione: in altre parole $S \rightarrow aSb$ (o anche

S::=aSb) significa che il simbolo non terminale *S* può essere derivato, sostituendo nella stringa le occorrenze di *S* con *aSb*. Nella grammatica del nuovo linguaggio implementato è stato usato il metasimbolo ::=.

Le categorie sintattiche sono generalmente rappresentate nella BNF racchiuse tra i simboli < e >, o scritte in grassetto. Si trovano quindi regole scritte in questi modo:

<*Espressione*> → <*Espressione*> + <*Espressione*>

Espressione ::= **Espressione** + **Espressione**

Dunque le regole scritte con la seguente notazione hanno una testa, a sinistra della freccia, generalmente scritta tra parentesi angolate, il metasimbolo → o il metasimbolo ::=, e il corpo della regola, costituito da zero o più categorie sintattiche e/o simboli terminali. È possibile raggruppare in un'unica produzione diversi corpi per la stessa testa, separandoli con il metasimbolo |, che possiamo leggere come “oppure”. Gli esempi di prima possono, dunque, essere scritti in tale modo:

$$S \rightarrow aSb \mid ab$$

2.4 Gli automi

Il secondo approccio per la definizione di un linguaggio è basato, come già detto, su degli strumenti chiamati automi. Tale strumento non si occupa di generare parole, come nel caso della grammatica e delle espressioni regolari, ma di identificare le stringhe che compongono il linguaggio. Questo si intende quando si parla di accettazione di un linguaggio da parte di una macchina.

Ogni tipo di linguaggio è riconosciuto da macchine differenti: i linguaggi regolari, ovvero quelli di tipo-3 secondo la classificazione di Chomsky, possono essere accettati da automi a stati finiti; i linguaggi liberi dal contesto (tipo-2) possono essere accettati dagli automi a pila; i linguaggi dipendenti dal contesto (tipo-1) possono essere accettati dagli automi lineari; i linguaggi illimitati (tipo-0) possono essere accettati da macchine di Turing.

Un automa è un metodo, basato su grafi, per specificare linguaggi; ci sono due tipi di automa, deterministico e non deterministico. Un automa non deterministico può essere trasformato in un automa deterministico che riconosce lo stesso linguaggio.

Un automa è in sostanza una macchina riconoscitrice di stringhe con la struttura di una macchina a stati: durante la lettura della stringa è possibile individuare dei punti in cui si riconoscono gli stati di avanzamento della macchina. Questi punti sono detti stati: la macchina funziona passando da uno stato a un altro, in conseguenza alla lettura dei dati, ovvero degli elementi che compongono la stringa.

Formalmente un automa è definito come una quintupla

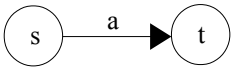
$$\langle Q, \Sigma, q_0, F, \delta \rangle$$

dove:

- Q è l'insieme finito degli stati dell'automa;
- Σ è l'alfabeto su cui è definito il linguaggio riconosciuto dall'automa;
- $q_0 \in Q$ è lo stato iniziale, appartenente all'insieme Q, in cui si trova l'automa quando inizia il tentativo di riconoscimento;
- $F \subseteq Q$ è l'insieme degli stati finali, sottoinsieme di Q;
- $\delta \subseteq Q \times \Sigma \times Q$ è la relazione di transizione che associa una coppia $(s, a) \in Q \times \Sigma$ a uno stato $s' \in Q$. L'appartenenza della coppia $\langle \langle s_i, a \rangle, s_j \rangle$ a δ significa che se l'automa si trova nello stato s_i e il simbolo da analizzare è a, allora si sposta sul simbolo successivo della stringa e nello stato s_j. La relazione di transizione δ è un sottoinsieme del prodotto cartesiano $Q \times \Sigma \times Q$.

Un automa si dice deterministico se da ogni stato non escono mai due transizioni etichettate con lo stesso simbolo.

È possibile rappresentare il funzionamento di una macchina a stati mediante un diagramma costituito da nodi e archi: i nodi rappresentano gli stati e graficamente sono raffigurati tramite cerchi; gli archi, invece, rappresentano le transizioni tra stato e stato e sono raffigurati da frecce che congiungono due nodi. Spesso i nodi e gli archi sono etichettati, ovvero è loro associata un'etichetta. Quando uno stato è uno stato di accettazione, o stato finale, è rappresentato da una doppia cerchiatura. Un automa che riconosce la relazione di transizione $\langle\langle s, a\rangle, t\rangle$, ad esempio, avrà una rappresentazione di questo tipo:



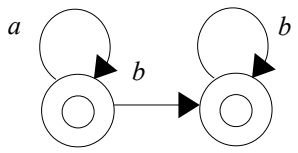
Formalizzando la nozione di accettazione, o riconoscimento, di una stringa da parte di un automa, diremo che un automa accetta (o riconosce) una stringa *s* se:

- esiste un cammino sul grafo dell'automa etichettato *s*;
- l'ultimo stato di tale cammino è uno stato finale.

Secondo questa definizione, un automa può fallire nel riconoscimento di una stringa per due motivi distinti: perché non riesce a completare l'esame della stringa per mancanza di opportune transizioni (archi) sul grafo, oppure perché l'esame della stringa termina in uno stato non finale.

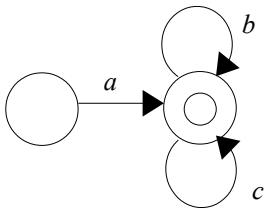
Il funzionamento di un automa è concettualmente semplice: dopo aver ricevuto una successione di caratteri, l'automa inizia dallo stato iniziale leggendo il primo carattere di tale sequenza. Sulla base del carattere esaminato viene eseguita una transizione ad un nuovo stato, che può coincidere con quello in cui l'automa già si trova. A questo punto l'automa legge il secondo carattere, esegue la transizione opportuna e così via.

Supponendo di avere un linguaggio regolare $L = \{a^* b^*\}$, ovvero un linguaggio le cui stringhe (o parole) sono formate da sequenze, anche vuote, di *a*, seguite da sequenze, anche vuote, di *b*, l'automa a stati finiti corrispondente sarà quello mostrato nel seguito:



Entrambi gli stati sono stati finali, in quanto il linguaggio presuppone l'accettazione della stringa vuota (ε).

Ancora, supponendo di avere un linguaggio regolare $L = \{a\ (b \mid c)^*\}$, ovvero un linguaggio le cui stringhe sono formate da una *a* seguita da una sequenza anche vuota di *b* o di *c*, l'automa a stati finiti corrispondente sarà quello mostrato nel seguito:



2.5 Le espressioni regolari

Le espressioni regolari sono il terzo metodo tramite il quale possiamo definire un linguaggio formale di tipo-3; sono, infatti, degli strumenti in grado di generare tutte e sole le stringhe che fanno parte del linguaggio.

L'insieme delle espressioni regolari su un certo alfabeto Σ è definito induttivamente dalle regole specificate nel seguito. Ogni espressione regolare su Σ definisce in maniera univoca un linguaggio su Σ. La definizione induttiva specifica (usando l'induzione sulla struttura delle espressioni) anche come viene formato il linguaggio definito da ogni espressione regolare:

Sia Σ un alfabeto finito. L'insieme delle espressioni regolari su Σ è costituito da tutte e sole le espressioni generate induttivamente come segue. Associamo ad ogni espressione anche il linguaggio denotato.

1. ε è un'espressione regolare che denota il linguaggio $\{\varepsilon\}$

2. Se a è un simbolo di Σ, allora a è un'espressione regolare che denota il linguaggio $\{a\}$

3. Siano r e s due espressioni regolari che denotano il linguaggi $L(r)$ e $L(s)$ rispettivamente. Allora:

    - $(r) \mid (s)$ è un'espressione regolare che denota il linguaggio $L(r) \cup L(s)$;

    - $(r)\,(s)$ è un'espressione regolare che denota il linguaggio $L(r)\,L(s)$;

    - $(r)^*$ è un'espressione regolare che denota il linguaggio $L(r)^*$;

- *(r)* è un'espressione regolare che denota il linguaggio *L(r)*.

Un linguaggio denotato da un'espressione regolare viene chiamato insieme regolare.

Di seguito alcuni esempi di espressioni regolari:

- sequenza di caratteri: i caratteri possono essere concatenati uno dopo l'altro

  $L = abc$

- stella di Kleene: denota una sequenza eventualmente di lunghezza pari a zero di caratteri contrassegnati con essa

  $L = a^*b = \{b, ab, aab, aaab, aaaaaab...\}$

- operatore + in apice al carattere: denota una sequenza di almeno un'occorrenza del carattere contrassegnato con esso

  $L = ab^+ = \{ab, abb, abbb, abbbbb...\}$

- scelta: indica una scelta tra due caratteri

  $L = a\ (a \mid b) = \{aa, ab\}$

- opzione: indica una o zero occorrenze del carattere contrassegnato con essa (i simboli che indicano l'operazione di opzione sono due: il carattere opzionale viene messo tra parentesi quadre o seguito da un punto interrogativo)

  $L = a[b]c = ab?c = \{abc, ac\}$

3 PRISM E IL MODEL CHECKING

Il model checking consiste in tecniche di verifica formale che consentono di controllare le proprietà comportamentali desiderate in un dato sistema, sulla base di un modello del sistema in questione, attraverso ispezioni sistematiche di tutti i suoi stati. L'attrattiva del model checking viene dal fatto che è completamente automatico e che offre controesempi nel caso in cui il modello fallisca nel soddisfare una proprietà che serve come informazione indispensabile per mettere a punto il sistema [16].

In questo capitolo verranno presentati tali tecniche, in particolar modo verranno esposti i modelli e i sistemi transizionali. Si rende, dunque, necessario spiegare che cosa si intende per modello e nello specifico che cos'è un modello di gioco, visto che è su questi ultimi che il lavoro di sperimentazione verte. Verrà poi presentato l'ambito di utilizzo del model checking. Infine, verrà mostrato lo strumento usato per compiere tali verifiche nel lavoro di sperimentazione da cui è nata l'idea per la stesura della tesi, PRISM.

3.1 I modelli e i sistemi transizionali

Un modello è un sistema di transizioni, il che significa che gli elementi che definiscono il modello in questione, possono assumere una serie di valori: tutti i possibili valori di tali elementi formano un insieme di stati del sistema; il passaggio da uno di questi stati a un altro è detto transizione. Un cammino è l'insieme delle configurazioni (degli stati) e delle transizioni che descrivono una possibile esecuzione.

Formalmente, dato un insieme di variabili X, un modello è definito da:

- l'insieme finito di stati S_x, che descrive tutti i possibili assegnamenti delle variabili;

- un insieme finito di transizioni che descrivono il passaggio da uno stato all'altro.

Ipotizzando di definire un modello per un sistema che gestisce l'invio e la ricezione di messaggi, si definiscono due elementi:

1. Il primo elemento del sistema, che verrà chiamato “host”, si occupa di inviare dei messaggi;

2. Il secondo elemento del sistema, che verrà chiamato “client”, si occupa di ricevere i messaggi;

Ogni elemento sarà descritto da una variabile che può assumere diversi valori, in particolare:

1. La variabile “host” può rappresentare l'aver inviato o meno il messaggio, dunque può assumere due valori differenti;

2. La variabile “client” può rappresentare l'aver ricevuto o meno il messaggio, e può rappresentare il volerne richiedere un altro, dunque può assumere tre valori differenti;

Secondo tali premesse il sistema può raggiungere un insieme di stati differenti:

1. La variabile “host” non ha inviato il messaggio.

2. La variabile “host” ha inviato il messaggio;

   La variabile “client” non lo ha ricevuto.

3. La variabile “host” ha inviato il messaggio;

   La variabile “client” lo ha ricevuto;

   La variabile “client” non ne richiede un altro.

4. La variabile “host” ha inviato il messaggio;

   La variabile “client” lo ha ricevuto;

   La variabile “client” ne richiede un altro.

Come si può notare il sistema è molto semplice, presenta solo due variabili, eppure può raggiungere un insieme di quattro stati differenti e a ogni stato possono corrispondere una o più possibili transizioni.

Il primo passo per verificare la correttezza di un sistema è quello di specificare le proprietà che tale sistema dovrebbe avere. Una volta riconosciute quali proprietà sono importanti e quali no, il secondo passo è quello di costruire un modello formale del sistema in questione. Per essere verificabile in maniera corretta ed efficiente, il modello deve catturare tutte le proprietà utili per stabilire la correttezza del modello. In altri termini, bisogna astrarre tutti quei dettagli che non concorrono nello stabilire la correttezza delle proprietà verificate ma che rendono la verifica più complicata. Ad esempio, se si vuole definire un modello per un protocollo di comunicazione bisogna focalizzarsi sullo scambio di messaggi e ignorare il reale contenuto di essi.

Per un corretto modello bisogna focalizzarsi sugli stati del sistema. Uno stato è una descrizione istantanea del sistema, che cattura i valori delle variabili in un determinato e particolare momento. È anche necessario sapere come lo stato del sistema cambia in risposta a determinate azioni: è possibile rappresentare il cambiamento descrivendo lo stato prima che l'azione si verifichi e dopo che l'azione si è verificata [17].

Una delle problematiche, ad esempio, che più vengono studiate attraverso la modellazione e il model checking è la *reachability*, ovvero la raggiungibilità di un determinato stato da parte di un sistema. Più nello specifico, la *probabilistic reachability*, che indica la probabilità che il sistema raggiunga quello stato. Formalmente, partendo da uno stato del modello S_x, si calcola la probabilità che il sistema stesso raggiunga un insieme di stati per i quali risulta vera una data proprietà.

Il concetto di *reachability* permette di indagare su due tipi di proprietà che un modello dovrebbe possedere per assicurare l'efficienza delle sue verifiche, le prime sono dette *safety properties*, le seconde sono dette *liveness properties*. Le *safety properties* per essere soddisfatte devono assicurare che durante i processi del sistema non si presenti un malus, ovvero che non si presenti nessuno stato corrispondente a una configurazione erronea; le *liveness properties*, invece, per essere soddisfatte devono assicurare che durante l'esecuzione, prima o poi, accada qualcosa di buono.

Un esempio del primo tipo di proprietà sono quelle che assicurano che un sistema non raggiunga mai una situazione di deadlock, ovvero una situazione di stallo in cui due o più processi si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro e viceversa. Supponendo di avere un modello di un gioco nel quale due giocatori, con tre punti ferita, lanciano un dado a testa: chi ottiene il risultato minore perde un punto ferita. Il sistema per andare avanti presuppone che:

- entrambi tirino il dado;

- chi ottiene il risultato minore perda un punto.

Qualora si arrivasse a un punto del gioco in cui uno dei due ha finito i punti ferita e, dopo il lancio del dado, ottenesse il risultato minore, il sistema non riuscirebbe a gestire la mancanza di punti ferita da sottrarre e si arriverebbe a una situazione di deadlock. Per evitare che questo si verifichi è necessario che il sistema sia in grado di gestire l'esaurimento dei punti ferita, per esempio tramite la terminazione della partita.

Garantire, invece, che un modello soddisfi una proprietà del secondo tipo, significa garantire un sistema in grado di progredire: ogni qualvolta che un giocatore ottiene un risultato maggiore dell'avversario è un evento positivo e consente al gioco di progredire.

L'adeguatezza di un modello spesso non è sufficiente per verificare se il sistema soddisfa o meno i requisiti che si vogliono testare. Potrebbe essere necessario sapere nel dettaglio quando il sistema li soddisfa e quando no.

Per poter rendere quantificabili i risultati di una verifica è possibile affidarsi a tecniche di model checking probabilistico o statistico. Nel caso di model checking probabilistico, i risultati si basano sul totale dei possibili cammini, ovvero sul totale effettivo delle possibili esecuzioni del sistema. Nel caso di model checking statistico, invece, i risultati si basano su una stima statistica: quando non è possibile verificare determinate proprietà, per esempio per la grandezza del sistema, è possibile effettuare le verifiche su un numero predefinito di cammini, tramite il metodo della simulazione – il model checker esegue un certo numero di cammini e verifica una determinata proprietà su di essi, restituendo una stima statistica. Il fenomeno descritto, ovvero la crescita del numero delle variabili di stato in un sistema e la conseguente crescita esponenziale del modello, che impedisce una corretta verifica delle proprietà, è chiamato *state explosion problem* [18].

3.2 Ambiti di utilizzo del model checking

Al giorno d'oggi i sistemi hardware e software sono largamente usati in ambiti in cui il fallimento è inaccettabile: solo per fare alcuni esempi, il commercio elettronico, le reti di scambio telefonico, i sistemi di controllo del traffico aereo e autostradale, gli strumenti medici. Spesso si legge di incidenti causati da errori nei sistemi software o hardware: si pensi al caso dell'esplosione del razzo Ariane 5, nel Giugno 1996. Anche quando gli errori non sono così fatali, come nel caso del Ariane 5, spesso possono essere

dannosi a livello economico – si pensi al caso del baco del Pentium 5 che causò ad Intel una perdita di 475 milioni di dollari. Inoltre, nella società in cui viviamo, grazie al successo di tale settore, diventa molto importante sviluppare metodi che possano incrementare la fiducia dei consumatori nella correttezza di questi sistemi, vista la crescente dipendenza dai sistemi computerizzati.

I metodi principali usati per la validazione di sistemi complessi sono:

- la simulazione;
- il *testing*;
- la verifica deduttiva;
- il model checking.

Sia la simulazione che il *testing* si basano sullo svolgimento di esperimenti prima della distribuzione del prodotto sul campo. La simulazione svolge esperimenti su un modello astratto del sistema, il *testing* invece sul prodotto reale, ma su insiemi selezionati di input e eventi esterni. Entrambi possono essere molto efficienti; tuttavia ambedue presentano dei problemi: la simulazione non può essere eseguita per sempre ed è spesso molto più lenta del sistema reale, inoltre, può essere molto costosa e non dare comunque nessuna garanzia che tutte le esecuzioni possibili vengano simulate; il *testing* soffre di svantaggi simili in quanto non tutte le configurazioni di input possono essere presentate al sistema: gli insiemi di input vengono generati automaticamente e non c'è nessuna garanzia che quelli "cattivi" vengano mai presentati. In sostanza la simulazione e il testing possono rivelare la presenza di bachi, ma non possono mai stabilirne l'assenza.

La verifica deduttiva, invece, si riferisce all'uso di assiomi e regole di dimostrazione per provare la correttezza del sistema. Il principale focus è quello di garantire la correttezza di sistemi critici e il comportamento corretto del sistema è assunto con così grande importanza che lo sviluppatore o un esperto di verifica (tipicamente un matematico o un logico) deve spendere tutto il tempo necessario per verificare il sistema stesso. Inizialmente queste regole di dimostrazione erano costruite interamente a mano, con gli anni sono stati creati degli strumenti automatici. L'importanza della verifica deduttiva è largamente riconosciuta nell'ambito della *Computer Science*; ha influenzato significativamente l'area dello sviluppo software. Tuttavia è un processo che richiede parecchio tempo, può essere fatta solo da esperti ed è molto difficile da automatizzare [17].

Il model checking, invece, come già detto, consiste nella costruzione di modelli e nella verifica formale di proprietà comportamentali desiderate, attraverso ispezioni sistematiche di tutti i suoi stati. Grazie a ispezioni di tutti gli stati del sistema, il model checking non ha il problema della simulazione e del *testing*, di non garantire l'assenza di bachi. Inoltre essendo completamente automatico, diventa più facile ed efficiente delle tecniche di verifica deduttiva.

Il model checking è stato usato per almeno due decadi quasi esclusivamente nell'ambito della *Computer Science*, soprattutto nel contesto di progetti hardware e software, settore dal quale nasce. Non è però difficile capire che tali tipi di descrizioni si adattano a molteplici tipi di sistemi: ad esempio, in campo medico, il corpo umano può essere visto come un sistema di variabili che possono assumere valori diversi e caratterizzare quindi stati differenti dello stesso.

In sostanza qualsiasi tipo di prodotto necessita in fase di progettazione o in fase di rilascio di tecniche per verificarne determinate caratteristiche e proprietà e, la modellazione di tali prodotti e la verifica formale dei modelli, non solo fa capire se effettivamente il prodotto soddisfa le cosiddette *liveness properties*, ovvero abbia delle caratteristiche che permettono al sistema di progredire, ma anche se soddisfa le cosiddette *safety properties*, ovvero controlla che il sistema non generi un errore.

I modelli e i sistemi transizionali si sono rivelati dunque perfetti per descrivere i giochi: sia perché i giochi sono, a tutti gli effetti, dei sistemi in cui i vari elementi (pedine, dadi, caselle del tabellone) possono essere descritti da variabili; sia perché l'analisi di tali modelli permette di verificare in maniera eccellente, diverse caratteristiche che, in fase di progettazione, il *game designer* desidera valutare per poter offrire una più soddisfacente esperienza di gioco.

Un modello di gioco corrisponde, dunque, a un sistema di transizioni, descritto da un insieme di variabili che rappresentano i vari elementi del gioco e da un insieme di transizioni che rappresentano i cambiamenti dei valori di tali elementi nel corso del gioco.

3.3 PRISM e il suo linguaggio

PRISM model checker [19] è uno strumento di modellazione e analisi di sistemi che, tramite l'utilizzo di alcune parole chiave, rende possibile la descrizione, oltre che del

modello stesso, delle proprietà che si intende far verificare al model checker. I modelli descritti da PRISM sono espressi come *discrete-time Markov chains* (DTMCs); le proprietà da verificare, invece, sono espresse tramite logica temporale.

A ogni stato del sistema possono corrispondere una o più possibile transizioni, che sono regolate da probabilità definite in fase di implementazione. Le transizioni possono essere associate a una data probabilità: se la transizione possibile è solo una, è legittimo omettere la probabilità da associarvi, che verrà impostata in automatico pari a uno; se invece è presente un set di transizioni, le singole probabilità devono sommare a uno, in caso contrario verrà segnalato un errore.

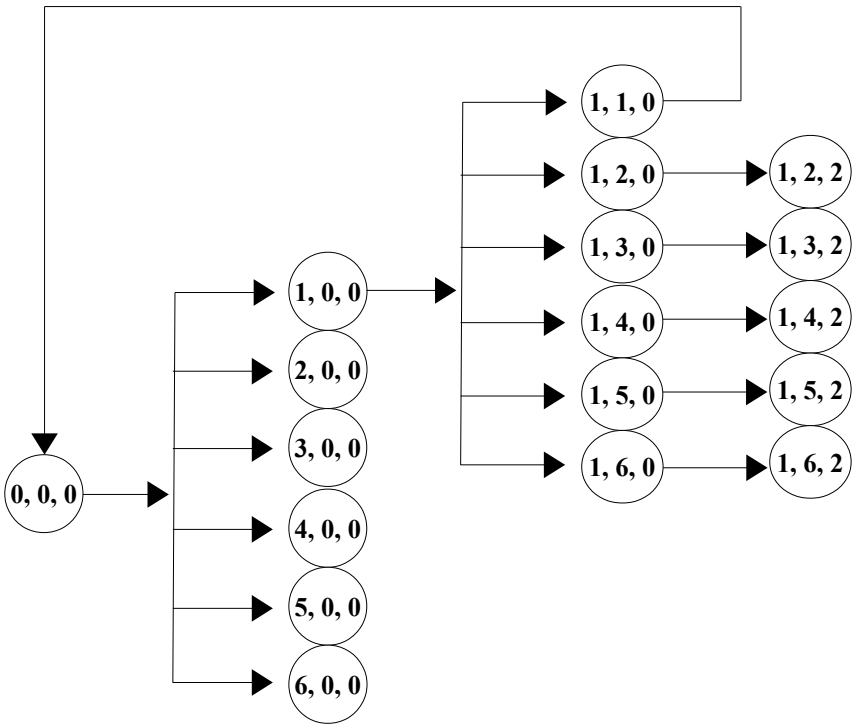
Di seguito la possibile implementazione di un modello che simuli un gioco molto semplice al fine di mostrare al meglio il linguaggio di PRISM model checker e mostrare come sono stati costruiti i modelli di gioco studiati in [3] (presenti nell'Appendice B).

```
dtmc

global dado1 : [0..6] init 0;
global dado2 : [0..6] init 0;
global vincitore : [0..2] init 0;

module Partita

[] (vincitore = 0) & (dado1 = 0) & (dado2 = 0) ->
                                                1/6 : (dado1' = 1) +
                                                1/6 : (dado1' = 2) +
                                                1/6 : (dado1' = 3) +
                                                1/6 : (dado1' = 4) +
                                                1/6 : (dado1' = 5) +
                                                1/6 : (dado1' = 6);

[] (vincitore = 0) & (dado1 != 0) & (dado2 = 0) ->
                                                1/6 : (dado2' = 1) +
                                                1/6 : (dado2' = 2) +
                                                1/6 : (dado2' = 3) +
                                                1/6 : (dado2' = 4) +
                                                1/6 : (dado2' = 5) +
                                                1/6 : (dado2' = 6);
```

```
[] (vincitore = 0) &
   (dado1 != 0) &
   (dado2 != 0) &
   (dado1 = dado2) ->
                    (vincitore' = 0) & (dado1' = 0) & (dado2' = 0);

[] (vincitore = 0) &
   (dado1 != 0) &
   (dado2 != 0) &
   (dado1 > dado2) ->
                      (vincitore' = 1);

[] (vincitore = 0) &
   (dado1 != 0) &
   (dado2 != 0) &
   (dado1 < dado2) ->
                      (vincitore' = 2);

endmodule
```

Il modello rappresenta un gioco dalle regole molto semplici: entrambi i giocatori possiedono un dado; il giocatore uno lancia il suo dado (`dado1`) e così fa anche il giocatore due (`dado2`); se il risultato è lo stesso si gioca nuovamente fintanto che uno dei due non ottiene un numero più alto dell'avversario.

Il model checker al momento della creazione del modello con le sue variabili e le sue transizioni è come se costruisse un grafo in cui i nodi rappresentano gli stati e le frecce rappresentano le transizioni. Ogni stato è rappresentato dai tre possibili valori delle tre variabili: le variabili `dado1` e `dado2`, che possono assumere valori da zero a sei e la variabile `vincitore` che può assumere valori da zero a due. Quando il gioco inizia tutte e tre le variabili hanno un valore pari a zero, quindi ci si trova nello stato *{0, 0, 0}*. Con 1/6 di probabilità la variabile `dado1` prende valore 1: ci si trova nello stato *{1, 0, 0}*. Con 1/6 di probabilità la variabile `dado2` prende valore 1: ci si trova nello stato *{1, 1, 0}*, dunque si ritorna allo stato iniziale. Se la variabile `dado2` avesse preso un valore pari a 2 ci si troverebbe in uno stato *{1, 2, 0}* che avrebbe portato allo stato *{1, 2, 2}* decretando la vittoria del giocatore due. Parte del grafo descritto viene mostrato di seguito: per una questione di spazio sono state mostrate solo gli stati in cui il `dado1` assume un valore pari a 1.



Il modello si apre con la dichiarazione del tipo di modello che si vuole creare: come già detto, tutti i modelli utilizzati nello studio sono Discrete Time Markov Chain (dtmc).

Gli elementi del sistema sono rappresentati da variabili: in questo caso ci sono due variabili `dado` che rappresentano i due giocatori e una variabile `vincitore`, inizializzata a zero e che si aggiorna a seconda di chi ottiene il risultato maggiore. Le tre variabili sono `global`, ovvero comuni a tutti il modello. Il linguaggio di PRISM prevede solo due tipi di variabili: interi e booleani. Le variabili vengono dichiarate nel seguente modo:

```
x: [0..2] init 0;
```

ovvero viene dichiarato il range dei possibili valori di x e viene inizializzata con la parola chiave `init`. È possibile anche non inizializzare la variabile o non specificare il range di valori.

In ogni modello è possibile specificare uno o più moduli che costituiscono il vero e

proprio motore del gioco, ovvero la parte del modello in cui vengono descritte le varie transizioni. In questo caso è presente il modulo Partita “aperto” con la parola chiave `module` e “chiuso” con la parola chiave `endmodule`.

All'interno del modulo vengono definite le varie transizioni del modello. In questo caso sono presenti cinque diverse transizioni che ricoprono tutti i possibili casi a cui si potrebbe andare incontro:

1. Lancio del primo giocatore;
2. Lancio del secondo giocatore;
3. Punteggio uguale per entrambi i giocatori;
4. Vincita del giocatore uno;
5. Vincita del giocatore due.

Le transizioni sono espresse nei seguenti modi:

```
[] (vincitore = 0) -> (vincitore' = 1)

   [] (dado1=0) -> 1/3: dado2'=1 +
                   1/3: dado2'=2 +
                   1/3: dado2'=3;
```

Come si può notare ogni transizione è composta da una guardia (tutto ciò che è scritto prima della freccia) che esprime una condizione sulla configurazione del sistema: se la guardia risulta vera allora il modello esegue gli aggiornamenti (tutto ciò che è scritto alla destra della freccia). Agli aggiornamenti possono essere o meno associate delle probabilità: come già detto, quando sono presenti delle probabilità devono sommare a uno (1/3 + 1/3 + 1/3 =1). Nel caso in cui, invece, la guardia risulta falsa, il modello ignora la transizione.

PRISM supporta anche l'uso delle costanti, che possono essere integer, double o booleani, e possono essere definiti usando la parola chiave `const`. PRISM permette di lasciare indefinito il valore di una costante e di impostarlo nella fase di esecuzione del model checking. In questo modo è possibile controllare, per esempio, il variare del risultato di una verifica al variare del valore di una data costante.

È inoltre possibile formulare delle espressioni: addizioni, sottrazioni, moltiplicazioni, divisioni, relazioni (<, <=, >, >=), uguaglianze e disuguaglianze (=, !=), negazioni, congiunzioni e disgiunzioni (&, |), espressioni condizionali (“condizione? A: B”, che significa, se la condizione è vera esegui A altrimenti esegui B).

Si possono anche utilizzare delle espressioni predefinite:

• min(...) e max(...), che selezionano rispettivamente il valore minimo e il valore massimo, di due o più numeri;

• floor(x) e ceil(x), che arrotondano rispettivamente all'intero inferiore e superiore più vicino;

• pow(x, y), che eleva x alla potenza di y;

• mod(i, n), che calcola il resto della divisione intera di i per n;

• log(x, b), che calcola il logaritmo di x in base b.

A seconda del tipo di modello e della funzione che questo è chiamato a simulare si renderà utile studiare una certa proprietà invece di un'altra. Anche le proprietà in PRISM hanno un loro specifico linguaggio: devono essere espresse tramite la logica temporale adeguata al tipo di modello. Nei modelli visti la logica temporale adottata prende il nome di *Probabilistic Computation Tree Logic* (PCTL) ed è l'estensione probabilistica della logica temporale CTL [16]. Gli operatori logici di maggiore interesse supportati dalla PCTL sono:

- P: uno dei più importanti operatori nel linguaggio delle proprietà, usato per verificare la probabilità che un determinato evento si compia

  ```
  P=? [ evento ]
  ```

  la probabilità può restituire un risultato numerico o un risultato booleano. È anche possibile verificare se la probabilità che l'evento si verifichi sia maggiore o minore di una certa percentuale (P>0.90 [vincitore=1], che verifica se la probabilità che vinca il giocatore uno sia maggiore del 90%).

- F: fa parte della logica temporale, l'operatore anche chiamato "Future", verifica con quale probabilità, partendo da un determinato stato, verrà raggiunto un altro

determinato stato:

```
P=? [F vincitore>0]
```

ovvero con quale probabilità è possibile raggiungere uno degli stati in cui un giocatore ha ottenuto un punteggio più alto dell'avversario aggiudicandosi così la vittoria.

- G: fa parte della logica temporale, l'operatore anche chiamato “Globally”, serve per esprimere l'eventualità che una certa proprietà sia valida in tutti gli stati del modello.

```
P=? [G(vincitore=0)]
```

ovvero con quale probabilità globalmente la variabile vincitore assume un valore pari a zero, non decretando mai un vincitore e che quindi il gioco non giunga a termine.

4 MODELLI DI GIOCO E IMPLEMENTAZIONE IN PRISM

Come già detto, il lavoro di sperimentazione svolto nasce dai lavori [2] e [3], nel corso del quale è stato usato il software PRISM model checker per definire dei modelli di gioco e per valutarne alcune caratteristiche.

Da questo lavoro ho ripreso i tre casi studiati; ad essi ne è stato aggiunto un quarto, sviluppato nell'ambito della mia tesi. I modelli di gioco sono dunque quattro e sono: il Gioco dell'Oca, un gioco di strategia ispirato al Risiko, un gioco del tipo dungeons crawl e un gioco ideato per acquisire dimestichezza con il software e il suo linguaggio, chiamato Lotta a Squadre.

4.1 Caratteristiche dei giochi

Se si pensa a un gioco da tavolo, lo si può facilmente descrivere come un sistema formato da diversi elementi: le pedine, i dadi, il tabellone, gli elementi aggiuntivi, solo per citarne alcuni. È naturale che gli elementi del gioco dipendono dal gioco stesso: un elemento aggiuntivo come per esempio le carte delle penalità nel Monopoli, non sono presenti in ogni tipo di gioco, ma resta il fatto che qualunque gioco ha degli elementi che lo contraddistinguono.

Un modello, dunque, è perfettamente adattabile a un gioco, anzi risulta essere un sistema ideale per la descrizione di esso: gli elementi appena descritti possono essere rappresentati perfettamente da variabili. Inoltre l'analisi dei modelli di gioco permette di verificare in maniera eccellente diverse caratteristiche che, in fase di progettazione, il *game designer* desidera valutare per poter offrire un'esperienza di gioco più soddisfacente.

Quali sono queste caratteristiche?

Come ci si può aspettare un gioco dovrebbe essere un'esperienza positiva: in generale si sceglie di giocare per divertirsi, o comunque per distrarsi e rilassarsi; si può anche giocare per il gusto per la competizione, con se stessi o con gli altri. Resta il fatto che

l'esperienza ludica è, o meglio dovrebbe essere sempre, un'esperienza piacevole per chi la compie. Perché sia un'esperienza di tale genere, il gioco deve, appunto, possedere determinate caratteristiche.

Un buon gioco in sostanza è un gioco che offre un'esperienza significativa, al punto che, chi ci ha giocato, deciderà di rifarlo. A tal proposito si parla di grado di *replayability* (letteralmente rigiocabilità) che un artefatto ludico è in grado di esprimere. Tra le caratteristiche che influenzano il giocatore nella scelta di un gioco, sia per il primo utilizzo che per i successivi, troviamo [3]:

- La durata del gioco e delle sue fasi:

  Il tempo speso a giocare deve essere sempre proporzionale alla complessità del gioco. Ci sono giochi in cui il tempo del turno è prestabilito, e spesso scandito da oggetti come timer e clessidre, e giochi in cui non ci sono vincoli. In entrambi i casi è importante che il tempo impiegato dai giocatori per portare a termine una partita sia ragionevole. A maggior ragione se si tratta di giochi in cui uno o più giocatori possono essere eliminati durante la partita: se il gioco durasse troppo per i giocatori eliminati sarebbe un'esperienza non di certo positiva.

- La varietà del gioco:

  Un gioco esprime la sua varietà quando offre scenari e ambientazioni alternativi o regole alternative che stravolgono la natura del gioco stesso. Un mazzo di carte, per esempio, a seconda delle regole che si intende seguire offrono una grande varietà di giochi possibili. In generale, delle regole e una struttura troppo rigide difficilmente permetteranno al gioco di esprimere un alto grado di varietà al suo interno.

- La randomizzazione della partita e delle sue fasi:

  Un gioco possiede questa caratteristica se gli obbiettivi, le risorse a disposizione o gli eventi sono assegnati in maniera casuale. Tipiche fonti di eventi casuali sono i dadi o un mazzo di carte dal quale i giocatori devono pescare per ricevere il proprio bonus o la propria penalità. Spesso questi elementi contraddistinti dalla *randomness* sono inseriti in giochi in cui è presente anche una logica di strategia, per lasciare spazio anche alle decisioni del giocatore.

- Delle differenti strategie vincenti:

Un gioco dovrebbe offrire al giocatore la possibilità di scegliere tra strategie differenti, che lo possano portare a raggiungere un determinato obbiettivo. Questa caratteristica si lega alla *replayability* in quanto il trovare una strategia vincente può rivelarsi un buon incentivo per rigiocare, nel tentativo di compiere delle scelte più efficaci. Questa caratteristica, inoltre, si lega a una struttura a "gradini" dei giochi, in particolar modo dei videogiochi, in cui l'esperienza diventa via via più difficile, man mano che si raggiungono degli obbiettivi intermedi.

- L'abilità di gioco cooperative/competitive:

  Mentre un gioco competitivo deve offrire al giocatore delle dinamiche che gestiscano gli scontri tra i giocatori, un gioco cooperativo (anche detto co-op) deve offrire ai giocatori delle meccaniche che mettano il giocatore in condizione di poter scegliere se collaborare o meno con un altro giocatore. Una delle problematiche più grandi legate ai giochi co-op, è l'uscita precoce dei giocatori dalla partita; tale problematica è presente anche nei giochi competitivi, ma in questo caso è il gioco che lo impone, dal momento che la vittoria di un giocatore si basa sull'uscita dal gioco di un altro giocatore. L'esperienza di gioco di chi viene eliminato verrà sicuramente influenzata negativamente, ma ciò avverrà anche per gli altri giocatori non eliminati. Altri due problemi legati a queste tipologie di gioco sono i cosiddetti *runway leader* e *leader bashing*: il primo designa la situazione in cui un giocatore ha creato un divario incolmabile con gli avversari, il secondo designa la situazione in cui il divario è recuperabile e i giocatori in svantaggio devono compiere azioni che rallentino la fuga del leader per poter vincere.

4.2 Proprietà dei modelli di gioco

Le cinque caratteristiche appena descritte devono essere tenute in considerazione da un *game designer* che vuole progettare un buon gioco e sono state tenute in considerazione nei modelli di gioco implementati in PRISM.

Il modo in cui vengono tenute in considerazione è quello di descrivere delle proprietà che siano in grado di valutarle correttamente. Di seguito verranno esposte le proprietà studiate nei casi studio del lavoro [3] e le proprietà studiate nel modello di gioco da me implementato.

Il primo modello è quello del Gioco dell'Oca, celeberrimo gioco da tavolo in cui due giocatori si sfidano lanciando un dado su un tabellone composto da 63 caselle e avanzano coprendo, di volta in volta, l'esatto numero di caselle indicato dal risultato del lancio del dado. Nel gioco originale sono presenti delle caselle "speciali": caselle penalità che riportano il giocatore indietro nel tabellone di un certo numero di posizioni e caselle che, invece, fanno andare avanti il giocatore. Per vincere bisogna arrivare per primi nella casella 63, se però con il risultato del lancio del dado si supera tale casella si deve ritornare indietro di un numero di caselle pari alla differenza tra il risultato del lancio del dado e le caselle che dividono il giocatore dall'ultima. In questo modello si è indagato su quale sia il numero di caselle penalizzanti tali da stabilizzare o meno le dinamiche di gioco. Le caselle penalizzanti rientrano a dovere in quegli elementi che regolano la *randomness* dei giochi.

Preliminarmente sono state fatte delle verifiche per capire dove fosse più opportuno posizionare tali tipi di caselle sul tabellone, arrivando alla conclusione che la posizione delle caselle penalizzanti non influisce sulla dinamica del gioco. In secondo luogo si è verificato quanto incide sulla vittoria di un giocatore l'andare a finire su una casella penalizzante, arrivando ovviamente al risultato che chi ci finisce più volte ha una probabilità minore di vincere. Infine si è verificato se esiste un numero ottimale di caselle che possa avere un effetto rilevante sullo svolgimento del gioco, arrivando al risultato che, superate le quattro caselle penalizzanti, si verificano le prime differenze rilevanti nello svolgimento del gioco: chi finisce meno volte su tali caselle ha una probabilità di vittoria raddoppiata rispetto a chi ci finisce di più. Dalle probabilità studiate si è arrivati alla conclusione che il numero ottimale di caselle penalizzanti è cinque, ovvero il numero minimo che ne massimizza l'effetto [3].

Il secondo modello si ispira al gioco del Risiko. Quest'ultimo è un gioco di strategia nel quale due o più avversari cercano di raggiungere l'obiettivo che gli è stato assegnato casualmente all'inizio della partita. Solitamente gli obiettivi consistono nel conquistare un certo numero di territori, distruggere le armate di un avversario, conquistare determinati continenti e così via. La plancia sulla quale si gioca si presenta come una mappa geopolitica della Terra nella quale sono considerati adiacenti sia gli stati che confinano naturalmente sia quelli che, anche se divisi per esempio da tratti di mare, sono comunque collegati da linee tratteggiate. Ogni giocatore, oltre a ricevere un obiettivo personale da perseguire, viene anche dotato di un certo numero di armate con le quali occupa i territori a lui assegnati in partenza e quelli che successivamente conquisterà giocando. Chi perde tutte le proprie armate viene eliminato dal gioco. Il gioco finisce quando uno dei giocatori raggiunge il suo obiettivo personale. In questo modello si è indagato sui possibili approcci alla partita e si osservato sia il variare della

durata del gioco, sia il variare della probabilità di vittoria al variare di tali approcci. Ovviamente questo si lega alla caratteristica, descritta precedentemente, per cui un gioco dovrebbe offrire ai giocatori differenti strategie vincenti, nonché alla caratteristica per cui un gioco dovrebbe tenere sotto controllo la durata delle sue partite.

Gli atteggiamenti studiati sono tre: aggressivo, ovvero l'atteggiamento di un giocatore molto propenso agli scontri con l'avversario, passivo, ovvero l'atteggiamento di un giocatore poco propenso agli scontri, e neutro, ovvero l'atteggiamento di un giocatore che alterna un comportamento aggressivo con quello passivo. Dalle proprietà verificate si è, ovviamente, riscontrato che un atteggiamento aggressivo comporti un accorciamento delle partite. Si è poi indagato su una possibile tattica predominante che sembra esistere ma costituisce una sorta di paradosso: la tattica predominante pare essere quella di non attaccare mai in modo da accumulare sempre più risorse (carri armati) e avere un livello di armate sufficiente a garantirsi la permanenza in gioco; tuttavia un atteggiamento di tal genere, se adottato da entrambi i giocatori, porterebbe a delle partite infinite e pertanto non ci sarebbe nessun vincitore. Allo stesso tempo, passare da una strategia passiva a una aggressiva pare essere molto remunerativo, soprattutto per chi fa trascorrere più turni senza attaccare. Dagli studi risulta, quindi, che non esiste una strategia dominante.

Il terzo modello è ispirato al gioco HeroQuest, nel quale quattro eroi devono sconfiggere le forze del male disposte sulla plancia di gioco da un master, ovvero un giocatore che interpreta il gioco stesso. Il master crea delle celle (*dungeons*) all'interno delle quali sono presenti queste forze del male che gli altri giocatori devono sconfiggere in maniera cooperativa. Nel modello si descrive un gioco con quattro giocatori che passano da una stanza all'altra. Le stanze in totale sono dieci: nelle prime nove vengono generati in maniera random da quattro a sette entità nemiche, che da qui in avanti chiamerò mostri, che i giocatori devono sconfiggere per passare alla stanza successiva; nella decima stanza i giocatori dovranno combattere con il mostro finale (MF). Ogni giocatore ha sette punti ferita: ogni qual volta viene ferito da un mostro ne perde uno, quando i punti ferita sono uguali a zero, il giocatore risulta morto, ovvero non può più partecipare al gioco. I combattimenti avvengono in questo modo:

- Combattimento tra un giocatore e un mostro (nelle prime nove stanze): un giocatore singolo tira due dadi e prende il risultato maggiore, il mostro ne tira solo uno: chi ha il punteggio maggiore vince; se è il giocatore a vincere il mostro viene immediatamente eliminato se invece è il mostro a vincere si procede con la sottrazione di un punto ferita del giocatore.

- Combattimento tra il MF e i giocatori rimasti in vita (nella stanza finale: i giocatori rimasti in vita lanciano due dadi, vengono sommati i punteggi maggiori, il MF lancia quattro dadi e vengono sommati i risultati: il punteggio maggiore vince.

La partita finisce quando i giocatori sconfiggono il MF o quando nessuno di essi rimane in vita. In tale modello di gioco si è affrontato il problema dell'eliminazione precoce di uno dei giocatori e si è indagato sulle proprietà che caratterizzano i giochi co-op, descritte precedentemente.

In questo caso studio si è tentato di definire dei parametri (punti ferita, numero di possibili bonus ai giocatori, e così via) che fossero in grado di garantire una migliore esperienza di gioco. Si è verificato come cambia il numero dei giocatori che sopravvivono per tutta la durata del gioco al variare dei punti ferita iniziali dati loro, tenendo anche conto del fattore incertezza: un gioco in cui si sa per certo che tutti arriveranno vivi alla fine della partita non è piacevole, così come uno in cui si viene eliminati troppo presto. Si è studiato, poi, come delle meccaniche soprannominate “delle cure”, ovvero che aumenta la cooperazione tra giocatori, influisce sulla possibilità che tutti i partecipanti sopravvivano più a lungo.

Il modello di gioco ideato per prendere dimestichezza con lo strumento PRISM model checker, il suo linguaggio e il linguaggio delle proprietà, è stato chiamato Lotta a Squadre. Il gioco è molto semplice: due squadre composte da tre giocatori l'una, giocano a turno: ogni giocatore possiede 10 punti ferita; se il giocatore di turno è ancora in vita (ovvero se possiede ancora un numero di punti ferita maggiore di zero), attacca un giocatore della squadra avversaria; gli attacchi vengono fatti con un semplice lancio di dado: il risultato del lancio del dado viene sottratto ai punti ferita dell'avversario. Sono stati implementati due modelli differenti. Nel primo modello la scelta su chi attaccare è fatta in maniera casuale: in questo caso è presente una forte componente random, sia perché il combattimento è fatto tramite il lancio di un dado che, come già detto, è uno degli elementi che regolano la *randomness* dei giochi, sia perché anche la scelta dell'avversario viene fatta in maniera casuale. Nel secondo modello, invece, la scelta su chi attaccare è regolata da strategie differenti: o si attacca sempre lo stesso giocatore finché non viene sconfitto o si attacca con probabilità variabile a seconda dei punti persi; in questo caso, non solo troviamo elementi che regolano la *randomness* dei giochi (si mantiene comunque il combattimento tramite i dadi), ma il modello si lega alla caratteristica dei giochi secondo la quale un artefatto ludico dovrebbe offrire differenti strategie ai giocatori.

Le proprietà studiate riguardano la probabilità di vittoria delle squadre per il modello random, ottenendo come risultato che le probabilità delle due squadre sono pressoché uguali, a parte il fatto che la squadra che inizia per prima a giocare ha una percentuale di poco più alta di vincere. Nel caso del modello con le strategie, è stato verificato quali tra i due atteggiamenti fosse il più appropriato, ottenendo, anche in questo caso, un risultato pressoché paritario, con una probabilità leggermente più alta per la strategia secondo la quale si attacca sempre lo stesso giocatore.

4.3 Casi studio implementati in PRISM

Come già detto, un modello di gioco corrisponde a un sistema di transizioni, descritto da un insieme di variabili che rappresentano i vari elementi del gioco e da un insieme di transizioni che rappresentano i cambiamenti dei valori di tali elementi nel corso del gioco.

Si è tenuto conto di questo nell'implementazione svolta in PRISM, creando dunque modelli in cui gli elementi del gioco sono variabili e i passaggi fatti per arrivare all'obbiettivo sono espressi da opportune transizioni. I quattro modelli sono presenti in allegato, di seguito verranno descritti nel dettaglio.

4.3.1 Gioco dell'Oca

Il codice sorgente di tale modello è consultabile in Appendice nella sezione B.1.

Nel modello implementato le caselle “bonus”, ovvero quelle che fanno avanzare il giocatore che vi capita, non sono state descritte: in prima istanza si è modellato un gioco nel cui tabellone sono presenti solo caselle “normali” e in seconda istanza si sono studiate le dinamiche di gioco con l'inserimento delle caselle penalità.

Il gioco, come tutti gli altri, è stato modellato come una DTMC (*Discrete Time Markov Chain*).

È definito un modulo per la gestione dei turni:

```
module MTurns
```

```
next_step : [1..5] init (2 * initial_player) - 1;
[turn1a] !terminated & next_step = 1 -> (next_step' = 2);
[turn1b] !terminated & next_step = 2 -> (next_step' = 3);
[turn2a] !terminated & next_step = 3 -> (next_step' = 4);
[turn2b] !terminated & next_step = 4 -> (next_step' = 1);
[termination] terminated -> (next_step'=5);
endmodule
```

Grazie alle labels (etichette) `turn1a, turn1b, turn2a, turn2b` è possibile regolare il lancio del dado da parte del giocatore uno e del giocatore due: il turno dei giocatori è diviso in due parti, nel turno a il giocatore lancia il dado, nel turno b il giocatore aggiorna la sua posizione e passa il turno all'avversario.

È poi definito un modulo per ciascun giocatore in cui i giocatori sono descritti dalle variabili *x* e *y*, ciascuno con il suo dado, *d1* e *d2*. È stato implementato un solo modulo e poi sincronizzato con la sostituzione letterale delle variabili, ovvero la sostituzione delle variabili che rappresentano gli stessi elementi: *y* con *x* e *d2* con *d1*.

L'azione è sincronizzata con il modulo del turno e prevede due transizioni:

1. Lancio del dado e aggiornamento della variabile giocatore (al suo interno viene gestita anche l'eventuale superamento della casella finale con il lancio del dado);

2. Gestione dell'occorrenza della casella penalità.

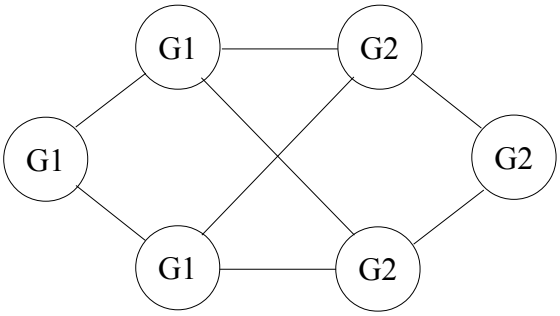
Le caselle penalità sono descritte come costanti, da definire volta per volta; lo stesso avviene per il numero di “passi indietro” da fare se si finisce in una casella penalità.

4.3.2 Risiko

Il codice sorgente di tale modello è consultabile in Appendice nella sezione B.2.

Nell'implementazione del modello il gioco è stato semplificato, ma ne è stata comunque mantenuta la meccanica principale, ovvero gli scontri tra avversari.

Il modello presenta una plancia formata da sei territori, collegati da 8 link come un grafo e due giocatori; si è deciso di fissare il numero di armate che è possibile spostare a uno. Di seguito il grafo che rappresenta la plancia di gioco.



Nel modello sono state già calcolate le probabilità di attacco e difesa, semplicemente simulando il lancio di due dadi, uno per l'attacco e uno per la difesa; la probabilità che vinca l'attacco è pari a 0,417, quella che vinca la difesa è pari a 0,583.

I territori sono definiti come variabili di tipo intero: se sono posseduti dal giocatore uno hanno valore 0, 1 altrimenti. Per ogni territorio sono presenti altrettante variabili per il conteggio delle armate. Non è presente un modulo per la gestione dei turni come nel modello del Gioco dell'Oca ma il passaggio del turno da un giocatore a un altro è fatto con l'aggiornamento della variabile `current_player`, fatto a seconda di determinate condizioni.

Il modello definisce una serie di formule per la gestione del gioco:

- formula di terminazione: se tutti i territori sono posseduti dallo stesso giocatore;
- formula per vedere se il numero di armate presenti in ogni territorio è maggiore di due;
- formula per calcolare il numero di territori posseduti da ogni giocatore (servirà per il rinforzo);
- formula per calcolare il numero di territori posseduti da ogni giocatore;
- formula per calcolare il numero di rinforzi;

- formula per vedere se si può attaccare un territorio: se è un territorio dell'altro giocatore e ha più di due armate;
- formula per calcolare il numero di territori attaccabili.

Il gioco è gestito da un unico modulo. All'inizio del modulo vengono inizializzate le variabili territorio (tre per ciascun giocatore) e le variabili armate (ogni territorio possiede 6 armate).

Sono definite sei fasi (dalla `phase 0` alla `phase 5)`:

1. Vengono distribuiti i rinforzi;
2. Si decide in maniera random se attaccare o meno: se si attacca si passa alla fase successiva, altrimenti si passa il turno all'altro giocatore;
3. Composta da due transizioni che si escludono a vicenda: se un territorio può essere attaccato si decide quale (ovvero si decide il territorio che si dovrà difendere), se nessun territorio può essere attaccato si passa il turno;
4. Viene stabilito da quale territorio sferrare l'attacco;
5. Per ogni coppia attacco/difesa si modellano i possibili risultati (con le probabilità già calcolate);
6. Se nessun territorio è privo di armate si continua a giocare (si torna alla prima fase ma senza nessun cambio di turno); se invece un territorio è vuoto si spostano le armate e si torna alla prima fase.

4.3.3 Dungeons crawl

Il codice sorgente di tale modello è consultabile in Appendice nella sezione B.3.

Anche in questo caso non è presente un modulo che gestisce i turni: è presente una variabile `current_player`, che gestisce il passaggio di "turno" da un giocatore a un altro (turno è scritto tra virgolette dal momento che in questo caso non esiste un vero e proprio turno: si parla più che altro di un passaggio di testimone da un giocatore a un

altro nel combattimento uno a uno con i mostri generati), e una variabile `room`, che gestisce il passaggio da una stanza a un'altra. A ogni giocatore è associata una variabile per il conteggio dei punti ferita, che parte da un numero massimo (in questo caso 7) a zero. Inoltre è definita una variabile `choice` per la scelta del giocatore di attaccare o non attaccare.

Una volta che i giocatori “entrano” in una stanza, quest'ultima viene inizializzata con la creazione dei mostri: con probabilità ¼ si creano 4, 5 , 6 o 7 mostri.

Tramite le formule che calcolano la probabilità di ogni giocatore di attaccare o meno (a seconda dei punti ferita posseduti) si stabilisce chi è il giocatore che sferra l'attacco, dopo di ciò si procede con lo stesso: anche in questo caso le probabilità che a vincere l'attacco siano i giocatori o i mostri è stato calcolato precedentemente. Se è il giocatore a vincere si aggiorna la variabile `current_player`, altrimenti si aggiornano i punti ferita. Infine viene descritto un modulo per gestire i diversi step presenti nell'ultima stanza e un modulo per la lotta finale nell'ultima stanza.

4.3.4 Lotta a squadre

Il codice sorgente di tale modello è consultabile in Appendice nella sezione B.4.

Come detto nel paragrafo precedente il gioco è stato ideato per acquisire dimestichezza con il linguaggio del model checker e sono stati definiti un modello in cui il gioco non è regolato da nessuna strategia ma solo dalla randomness, e uno in cui il gioco è regolato da strategie, per simulare degli ipotetici approcci di giocatori reali: attaccare sempre lo stesso giocatore fino a quando questo non esaurisce i suoi punti ferita o attaccare con probabilità diverse a seconda dei punti ferita posseduti. Di seguito verrà spiegata l'implementazione del secondo modello, quello con strategie.

È stato definito un modulo per la gestione dei turni, e in questo caso è stata definita una costante `giocatoreIniziale`, per capire se la vittoria di una squadra piuttosto che di un'altra è dovuta a una migliore strategia o semplicemente al fatto che abbia iniziato a giocare per prima o meno.

Sono stati definiti due moduli distinti: il modulo `MSquadra1` che utilizza la strategia secondo la quale gli attacchi sono sferrati con probabilità diverse a seconda dei punti ferita degli avversari; il modulo `MSquadra2`, invece, utilizza la strategia secondo la quale si attacca sempre lo stesso giocatore finché non ha esaurito i punti feriti.

Il modulo `MSquadra1` ha al suo interno tre variabili (*a, b, c*) che rappresentano i giocatori, o meglio i punti ferita dei giocatori, della squadra avversaria: inizializzate a dieci. Le transizioni di tale modulo usano le seguenti formule, implementate all'inizio del modello:

- `tuttiUguali1`: la formula controlla che i giocatori della squadra avversaria siano in vita e abbiano gli stessi punti ferita; nel caso i giocatori abbiano tutti gli stessi punti ferita la transizione viene effettuata con probabilità 1/3 per ogni giocatore.

- `attaccoA, attaccoB, attaccoC`: se i punti persi dai giocatori della squadra avversaria sono pari a dieci, restituisce 0, altrimenti restituisce i punti persi + 5: il motivo è quello di pesare le perdite dei punti ferita da parte dei giocatori in modo da avere delle probabilità diversificate a seconda di questi ultimi; le probabilità di attacco infatti saranno calcolate come il risultato di tale formula fratto la somma dei risultati delle tre formule (una per ciascun giocatore);

Il modulo `MSquadra2`, invece, ha al suo interno le variabili *x*, *y*, *z*, corrispondenti ai giocatori (punti ferita dei giocatori) della squadra avversaria. Le transizioni di questo modulo usano le seguenti formule:

- `tuttiUguali2`: uguale a tuttiUguali1 ma con le sostituzioni delle variabili dell'altra squadra; nel caso in cui i giocatori abbiano tutti gli stessi punti ferita la transizione viene eseguita con una probabilità calcolata come il risultato delle formule xVivo, yVivo, zVivo fratto il numero dei giocatori vivi;

- `PerdenteX, PerdenteY, PerdenteZ`: restituisce il numero dei giocatori vivi se il giocatore in questione è effettivamente quello con meno punti, zero altrimenti; utilizzate per calcolare le probabilità di attacco: il risultato di tali formule fratto il numero dei vivi;

- `NumVivi`: restituisce il numero dei giocatori vivi;

- `xVivo, yVivo, zVivo`: restituisce uno se il giocatore è vivo, zero altrimenti.

Il modello definisce anche la costante k, un intero da inserire al momento della simulazione che rappresenta il numero di facce del dado; tale variabile è stata inserita per verificare se le dinamiche di gioco cambiano a seconda delle facce del dado.

5 I REQUISITI DI UN DOMAIN SPECIFIC LANGUAGE PER LA MODELLAZIONE DI GIOCHI

In tutti i settori scientifici e ingegneristici si può distinguere tra approcci generici e approcci specifici. Un approccio generico fornisce una soluzione generale per molti problemi in una determinata area. Un approccio specifico, invece, fornisce soluzioni migliori per un set più limitato di problemi.

I vecchi linguaggi di programmazione nascono tutti come linguaggi dedicati a risolvere problemi in una determinata area: Cobol nato nell'ambito del commercio bancario, Fortran nato come traduttore di formule matematiche in algoritmi computazionali, Lisp nato per studiare le equazioni di ricorsione in un modello computazionale. Gradualmente si sono evoluti in linguaggi con propositi generali e pian piano il bisogno di linguaggi più specializzati, in grado di risolvere problemi in domini ben definiti è riemerso. Nel tempo sono state provate le seguenti soluzioni [20]:

- Librerie di sottoprogrammi: eseguono attività correlate in domini ben definiti, come per esempio equazioni differenziali, grafici, interfacce utenti e database. Le librerie di sottoprogrammi sono dei metodi classici per questi settori di conoscenza.

- Programmazione Orientata agli aspetti (AOP): nasce a partire dalla Programmazione Orientata agli Oggetti (OOP): gli aspetti modellano problematiche trasversali agli oggetti stessi, ossia compiti che nell'OOP tradizionale sono difficilmente modellabili.

- Domain Specific Language (DSL): è un linguaggio di programmazione o di specifica dedicato a particolari problemi di dominio, a una particolare tecnica di rappresentazione e/o una particolare soluzione tecnica. Spesso i programmi scritti con un DSL sono tradotti per chiamare una comune libreria di sottoprogrammi e il DSL può essere visto come un metodo per nascondere i dettagli di tale libreria.

Sebbene molti DSL sono stati progettati e usati da anni, lo studio sistematico di questo tipo di linguaggi è iniziato solo recentemente. I Domain Specific Language sono spesso

linguaggi limitati che offrono solo un set ristretto di notazioni e astrazioni. In letteratura sono anche chiamati micro-linguaggi.

Nel lavoro di sperimentazione da me svolto un tale tipo di linguaggio è risultato la soluzione ottimale per riuscire a superare le problematiche che si sono riscontrate con un linguaggio come quello di PRISM.

5.1 Vantaggi e svantaggi dell'implementazione nel linguaggio di PRISM

Come già espresso più volte, le tecniche di verifica formale e in particolare il model checking offrono un approccio potente e rigoroso per stabilire la correttezza dei sistemi complessi. I miglioramenti nell'efficienza e nell'usabilità di queste tecniche, che negli anni sono stati fatti, hanno portato a utilizzarle nella fase di progettazione di un ampio insieme di sistemi. I sistemi in questione possono essere non solamente sistemi computerizzati, dato che una grande quantità di prodotti (prodotto inteso come bene o servizio) possono essere visti come un sistema costituito da stati e transizioni tra questi ultimi.

Il model checking probabilistico è una generalizzazione di queste tecniche focalizzato su sistemi il cui comportamento è stocastico, ovvero sistemi in cui le transizioni da uno stato a un altro variano in base a leggi probabilistiche. Il model checking probabilistico è basato sulla costruzione e l'analisi di un modello probabilistico, tipicamente questi modelli sono espressi come catene di Markov o processi markoviani.

Una catena di Markov (*Markov chain*) è un processo random nel quale la probabilità di transizione che determina il passaggio ad uno stato di sistema dipende unicamente dallo stato di sistema immediatamente precedente (tale caratteristica è detta proprietà di Markov) e non da come si è giunti a tale stato.

Un processo markoviano (*Markov decision process*) è una struttura matematica per modellare processi decisionali in situazioni in cui i risultati sono in parte casuali e in parte sotto il controllo di un sistema che prende decisioni.

Il model checking probabilistico richiede due input:

1. una descrizione del sistema che deve essere analizzato, tipicamente fatta in un

linguaggio di modellazione di alto livello;

2. una specificazione formale di proprietà quantitative del sistema che devono essere analizzate, solitamente espresse in varianti di logica temporale.

Dal primo di questi input un model checker probabilistico costruisce il corrisponde modello probabilistico, che sarebbe una variante probabilistica di un sistema di stati-transizioni: ogni stato rappresenta una possibile configurazione del sistema che è stato modellato e ogni transizione rappresenta una possibile evoluzione del sistema da una configurazione a un'altra.

Il potere del model checking probabilistico viene dal fatto che questi modelli sono costruiti in maniera esaustiva, basata su esplorazione sistematica di tutti i possibili stati che possono verificarsi. Una volta che questo modello è stato costruito, può essere usato per analizzare un ampio range di proprietà quantitative del sistema originale che hanno a che fare, per esempio, con la sua prestazione o la sua affidabilità.

Sono stati sviluppati negli anni molti formalismi per descrivere modelli probabilistici, ma PRISM fornisce un linguaggio di modellazione semplice e testuale [21].

PRISM è un model checker probabilistico open-source sviluppato inizialmente all'Università di Birmingham e in seguito all'Università di Oxford. Tale strumento fornisce supporto per la costruzione e l'analisi di diversi tipi di modelli probabilistici: *Discrete Time Markov Chain* (DTMC), *Continuous Time Markov Chain* (CTMC), *Markov Decision Process* (MDP), *Probabilistic Timed Automata* (PTAs).

Questi modelli possono essere utilizzati dal model checker descrivendoli nel linguaggio di PRISM, un linguaggio di modellazione di alto livello pressoché semplice.

Le proprietà di tali modelli, invece, sono scritte con un linguaggio di PRISM specifico per esse, basato sulla logica temporale. Tale linguaggio incorpora diverse logiche temporali probabilistiche ben note: *Probabilistic Computation Tree Logic* (PCTL), *Continuous Stochastic Logic* (CSL), *Linear Time Logic* (LTL), PCTL* (che ingloba PCTL e LTL).

PRISM model checker è stato usato per analizzare un ampio insieme di casi studio in molti ambiti di impiego [22]:

1. Algoritmi di distribuzione random: sono stati sviluppati diversi casi studio che

hanno esaminato la correttezza e la prestazione di diversi algoritmi con distribuzione random. I casi studiati sono molto vari, di seguito alcuni esempi:

- Il problema dei filosofi a cena [23], altrimenti noto come problema dei cinque filosofi: è un esempio che illustra un comune problema di controllo della concorrenza in informatica, in sostanza un problema di sincronizzazione fra processi paralleli. Il problema consiste nello sviluppo di un algoritmo che impedisca lo stato di deadlock (stallo) o lo stato di *starvation* (indedia). La presa di forchette è analoga al blocco di risorse limitate nella programmazione, situazione nota con il nome di concorrenza. Bloccare una risorsa è una tecnica comune per garantire l'accesso da parte di un programma solo o di una sola porzione di programma alla volta. Se il blocco coinvolge più di una risorsa si può verificare una situazione di deadlock.

- Problema dei generali bizantini: è un problema informatico su come raggiungere consenso in situazione in cui è possibile la presenza di errori. Il problema consiste nel trovare un accordo, comunicando solo tramite messaggi, tra componenti diversi nel caso in cui siano presenti informazioni discordanti.

- Programmi di dadi: questo caso studio considera e riformula il lavoro di D. Knuth e A. Yao, "The complexity of nonuniform random number generation."

2. Protocolli di comunicazione, di rete e multimediali: i seguenti casi studio investigano su proprietà come la qualità dei servizi di tali protocolli, in particolare sono stati implementati modelli per casi studio quali:

   - Protocolli di comunicazione per dispositivi bluetooth;

   - Protocolli di trasmissione probabilistica;

   - Protocolli Gossip, ovvero la classe di protocolli di comunicazione

     ispirata al modo in cui il pettegolezzo diffonde messaggi nelle reti sociali, ovvero disseminando contenuti attraverso una rete, basandosi su scambi periodici di dati con membri casuali della rete.

3. Sicurezza: casi studio legati a sistemi che hanno a che fare con la sicurezza, ad esempio:
    - Attacchi agli schemi dei PIN;
    - Virus alle reti informatiche;
    - Protocolli di scambi equi e sicuri.
4. Biologia:
    - Controllo del ciclo della cellula negli Eucarioti;
    - Studi sul DNA;
    - Semplici reazioni molecolari.
5. Teoria dei giochi
6. Gestione energetica

Come si può notare PRISM è stato usato per analizzare la prestazione e l'affidabilità di casi studio tra un ampio insieme di provenienza: sistemi dinamici di gestione della potenza, sistemi di controllo, circuiti, protocolli di comunicazione, reti di computer, giochi, processi biologici e sistemi di produzione, per citarne solo alcuni. Il sito ufficiale dello strumento di model checking[2] presenta una vasta lista di lavori svolti con diversi esempi e la bibliografia delle pubblicazioni.

Appare dunque chiaro come PRISM model checker sia uno strumento ricco e utile in diversi ambiti e campi del sapere. A tutti questi vantaggi si accompagnano, tuttavia, dei problemi legati principalmente alla semplicità del linguaggio con cui vengono descritti i modelli.

Come già descritto nel terzo paragrafo del capitolo tre (“PRISM e il suo linguaggio”), il linguaggio di PRISM è pressoché semplice. Il linguaggio presenta:

1. Variabili locali di tipo integer o boolean

2 http://www.prismmodelchecker.org/

```
x: [0..2] init 0

y: bool init false;
```

2. Variabili globali (comuni a tutto il modello)

```
global g: [1..10];
```

3. Costanti di tipo integer, double o boolean (possono essere inizializzate o inizializzarle in fase di model checking, in modo da poter verificare determinate proprietà a seconda del cambiamento di una data costante)

```
const integer b;

const double pi_greco = 3.141592;

const bool yes = true;
```

4. Transizioni composte da una guardia a sinistra della freccia e uno o più aggiornamenti a destra della freccia eventualmente associati a una probabilità che deve sommare sempre a uno

```
[] x > 2 -> 1/3: x'=0 +
            1/3: x'=1 +
            1/3: y'=0;
```

5. Espressioni: addizioni, sottrazioni, moltiplicazioni, divisioni, relazioni, uguaglianze e disuguaglianze, negazioni, congiunzioni e disgiunzioni, espressioni condizionali;

6. Espressioni predefinite: min, max, floor, ceil, pow, log.

7. Rewards (ancora in fase di implementazione in PRISM): possono essere verificati con le proprietà, per esempio, nel conteggio dei turni

```
rewards
  [turno1] true: 1;
  [turno2] ture: 1;
endrewards
```

8. Formule

```
formula squadra_wins = a=0 & b=0 & c=0;
```

Come si può notare, il linguaggio è praticamente privo di strutture dati: a parte le singole variabili e le costanti non sono presenti altri tipi di rappresentazione dei dati. I costrutti di programmazione, inoltre, sono praticamente assenti: i comandi potrebbero essere visti come comandi condizionali del tipo if-else-then, ma nell'utilizzarli ci si rende subito conto che sono molto più complicati; non esistono costrutti che permettono di iterare una determinata azione.

A ciò si aggiunge, nel nostro caso specifico, che un linguaggio di tal genere è inadeguato a descrivere in maniera concisa alcuni aspetti fondamentali dei giochi: le strategie dei giocatori, l'ambiente di gioco (come ad esempio il tabellone di un gioco da tavolo) e le scelte probabilistiche con distribuzioni legate allo stato del gioco.

Tutto ciò ha reso piuttosto complessa la modellazione dei casi modellati e studiati in [3], nonché il quarto caso modellato nell'ambito del mio lavoro.

Andando a valutare i modelli svolti – allegati nell'Appendice B– si nota subito come la mole di codice scritto sia molto ampia, anche laddove si vuole gestire eventi del gioco relativamente semplici; vien da sé che con un linguaggio di programmazione più ricco certi passaggi diventerebbero decisamente più semplici e snelli.

Prendiamo, ad esempio, in considerazione il modello del gioco del Risiko (consultabile nella sezione B.2). In questo modello abbiamo:

- sei diverse variabili di tipo intero (`s1, s2, s3, s4, s5, s6`) che rappresentano i sei diversi territori, inizializzate a 0 o a 1 a seconda che il territorio sia di proprietà del giocatore uno o del giocatore due;

- sei diverse variabili di tipo intero per memorizzare il numero di carri armati di ciascun territorio (`c1, c2, c3, c4, c5, c6`);

- sei diverse formule per valutare se il giocatore corrente (`current_player`) ha almeno due carri armati nei territori di sua proprietà:

```
formula cs1 = (s1 = current_player & c1 >= 2) ? 1 : 0;
formula cs2 = (s2 = current_player & c2 >= 2) ? 1 : 0;
```

```
formula cs3 = (s3 = current_player & c3 >= 2) ? 1 : 0;
formula cs4 = (s4 = current_player & c4 >= 2) ? 1 : 0;
formula cs5 = (s5 = current_player & c5 >= 2) ? 1 : 0;
formula cs6 = (s6 = current_player & c6 >= 2) ? 1 : 0;
```

laddove il territorio sarà di proprietà del giocatore corrente e avrà più di due carri armati l'espressione condizionale restituirà il numero 1, altrimenti restituirà 0.

- Altre sei formule per valutare se l'avversario del giocatore corrente possiede un territorio:

```
formula other_player_owns_s1 = 1 - current_player_owns_s1;
formula other_player_owns_s2 = 1 - current_player_owns_s2;
formula other_player_owns_s3 = 1 - current_player_owns_s3;
formula other_player_owns_s4 = 1 - current_player_owns_s4;
formula other_player_owns_s5 = 1 – current_player_owns_s5;
formula other_player_owns_s6 = 1 - current_player_owns_s6;
```

- A loro volta i risultati delle ultime due formule descritte verranno utilizzati in altrettante sei formule che valuteranno se un territorio può attaccarne un altro o meno.

```
formula can_attack_s1 = other_player_owns_s1 * max(cs2, cs3);
formula can_attack_s2 = other_player_owns_s2 * max(cs1, cs4,
                        cs5) ;
formula can_attack_s3 = other_player_owns_s3 * max(cs1, cs4,
                        cs5);
formula can_attack_s4 = other_player_owns_s4 * max(cs2, cs3,
                        cs6);
formula can_attack_s5 = other_player_owns_s5 * max(cs2, cs3,
                        cs6);
formula can_attack_s6 = other_player_owns_s6 * max(cs4, cs5);
```

Ci si renderà subito conto che sono tante righe di codice solamente per verificare se un territorio può attaccarne un altro.

È vero che si arriva comunque ad ottenere il risultato desiderato, ma non è difficile pensare che con un linguaggio di programmazione più ricco il modello sarebbe

enormemente semplificato.

Supponiamo, ad esempio, di programmare il modello del gioco del Risiko in Java. Supponendo di avere una classe Territorio: ogni oggetto Territorio avrà al suo interno una variabile che ne memorizza il numero di armate e una variabile che ne memorizza il proprietario. Ancora, supponendo di inserire all'interno di un Vettore territori tutti gli oggetti Territorio e di avere un vettore `dif`, che raccoglie i territori attaccabili (ovvero chi si dovrà difendere nell'attacco), con un semplice ciclo for e un comando condizionale if potrei esprimere in poche righe ciò che, con il linguaggio di PRISM, è stato espresso in almeno trenta righe di codice:

```
for(int i=0; i< territori.length; i++){
   if(terrritori[i].proprietario!=current_player & t[i].armate>2){
      dif.add(territori[i]);
   }
}
```

5.2 Vantaggi di un nuovo linguaggio di modellazione

Da questi presupposti nasce quindi l'idea che sta alla base del lavoro di sperimentazione svolto, di creare un nuovo linguaggio di modellazione di alto livello, che renda più semplice e rapida la costruzione di modelli di gioco. Un Domain Specific Language pare, quindi, la soluzione ideale.

Nell'arrivare a tale decisione sono state fatte preliminarmente delle classi Java per ogni modello di gioco, nel tentativo di decidere se utilizzare tale linguaggio di programmazione di alto livello. Ci si è, tuttavia, resi subito conto che numerosi costrutti e comandi di Java risultano superflui per la modellazione di giochi, di fatti i modelli implementati in Java risultavano, seppure meno di quelli implementati nel linguaggio di PRISM, poco concisi.

I vantaggi che si possono riscontrare nella creazione di un nuovo linguaggio riguardano principalmente la ricchezza di tale linguaggio, che quindi porterebbe a una modellazione di programmi più semplici e snelli, dotati di strutture dati più ricche, di metodi ad hoc per le varie esigenze di gioco, di comandi condizionali e di costrutti che permettano di iterare delle azioni.

Inoltre un linguaggio di questo genere sarebbe enormemente più adeguato a descrivere alcuni aspetti fondamentali dei giochi:

1. le strategie dei giocatori;
2. l'ambiente di gioco;
3. la turnazione.

Andiamo ad analizzare tali aspetti, tenendo conto dei quattro diversi modelli.

5.2.1 La strategia dei giocatori

Quello che rende un modello di gioco ancora più vicino al comportamento al sistema reale, ovvero il gioco reale, è appunto la strategia. Un modello in cui facilmente sono esprimibili diverse strategie di gioco descrivono un'azione molto vicina a quella di un giocatore reale: chi gioca a Risiko, ad esempio, non avrà uno schema prestabilito per decidere se attaccare o meno ma valuterà a seconda della situazione in cui si trova, alla situazione in cui si trova l'avversario, all'obbiettivo che deve raggiungere e, perché no, anche allo stato d'animo in cui si trova. La decisione di utilizzare una strategia piuttosto che un'altra, dunque, varia di partita in partita, se non anche di turno in turno.

Avere un linguaggio che riesce, con più facilità, a simulare un atteggiamento “umano” nella scelta della strategia, comporterà la scrittura di modelli sempre più affidabili e più precisi nell'esame delle proprietà che si vuole verificare.

Il Gioco dell'Oca, come già detto, è un gioco basato essenzialmente sull'alea. I giocatori si limitano a tirare un dado e non è, quindi, presente alcun tipo di strategia.

Il Risiko, invece, come più volte espresso, fa di tale caratteristica la sua forza. Come visto nel paragrafo precedente, con il linguaggio di PRISM risulta quantomeno lungo, ma anche complicato, anche solo scegliere quale territorio attaccare. Ipotizzando, invece, di avere un linguaggio con strutture dati più complesse, come ad esempio dei grafi, e con dei metodi studiati ad hoc, lo scegliere chi attaccare e soprattutto quando farlo, risulterebbe enormemente più semplice.

Lo stesso avviene per il gioco collaborativo. In questo contesto la strategia del giocatore si limita a valutare se aiutare gli altri giocatori con i propri punti ferita in modo da

riuscire ad arrivare alla stanza finale con più giocatori in vita possibili, oppure tenere i propri punti ferita in modo da avere più possibilità di arrivare vivo fino all'ultima stanza. Anche in questo caso si ipotizza di avere dei metodi adeguati per la scelta: metodi che magari tengono conto di quanti sono ancora i giocatori in vita o che si basano sul numero di punti ferita del giocatore stesso (superata una certa soglia si decide di non "curare" gli altri giocatori), facilmente esprimibili con dei costrutti condizionali.

Tutto questo diventa ancora più chiaro per il quarto modello di gioco. Ipotizzando di costruire le squadre e i giocatori come delle classi, si potrebbe pensare di implementare metodi per le due diverse strategie.

5.2.2 L'ambiente di gioco

Lo studioso olandese J. Huizinga, autore di [24], nell'opera che ha contribuito alla nascita dell'interesse scientifico per il mondo dei giochi, si sofferma sulla separazione spazio-temporale che caratterizza il gioco, definendo il luogo destinato al gioco un cerchio magico. Chi gioca sa perfettamente quali sono i confini del proprio agire, dove inizia il "terreno" di gioco e dove finisce. Nel caso di un perimetro tracciato per terra, per esempio in un campo di pallacanestro, è facilmente riconoscibile (non solo per chi gioca, ma anche per chi assiste come spettatore) quali sono le delimitazioni che racchiudono lo spazio di gioco; lo stesso può valere per un gioco da tavola che preveda un tabellone sul quale disporre e muovere le pedine dei giocatori [3].

Da questi presupposti si può ben capire quanto sia importante, soprattutto nel caso dei giochi da tavolo, l'ambiente di gioco.

Nel Gioco dell'Oca abbiamo un tabellone formato da 63 caselle: sebbene sia praticamente uno degli elementi caratterizzanti di tale gioco, le caselle possono essere facilmente descritte come variabili. Di fatto in questo caso il tabellone non risultava essere problematico neanche nel modello implementato nel linguaggio di PRISM, nel quale sono le stesse variabili giocatore a memorizzare la posizione che via via occupano. Nel nuovo linguaggio si può pensare di avere una classe giocatore con al suo interno una variabile casella che viene aggiornata a ogni turno.

Le cose si complicano per quel che riguarda il modello del Risiko: è vero che la plancia di gioco parrebbe abbastanza semplice, essendo formata da soli sei territori; ma è anche vero che, tali territori, sono collegati tra loro da otto link. Con delle strutture dati semplici come le variabili, risulta difficile descrivere un tale tipo di tabellone. Con un

nuovo linguaggio creato ad hoc, si potrebbe pensare di creare delle strutture dati più ricche ma non eccessivamente complesse: ad esempio creando dei grafi i cui elementi sono, appunto, i link tra territori.

Anche nel caso del modello di Dungeon Crawl il tabellone di gioco risulta essere più complicato da descrivere con il linguaggio di PRISM, soprattutto a fronte del fatto che, in ogni stanza, viene generato un numero di mostri variabile da quattro a sette. Con il linguaggio di programmazione ipotizzato sinora, tuttavia, il creare degli oggetti stanze con all'interno degli oggetti mostri che vengono creati in maniera random in numero variabile, l'ambiente di gioco risulterebbe enormemente semplificato.

L'ultimo modello non presenta problemi da questo punto di vista, non avendo di fatto un tabellone di gioco che influenza le partite e le azioni dei giocatori.

5.2.3 La turnazione

Uno degli aspetti fondamentali dei giochi è lo scambio di turni tra giocatori avversari.

Nel Gioco dell'Oca la turnazione viene svolta tramite un modulo apposito e un sistema di sincronizzazione delle etichette. Ogni giocatore gestisce due fasi della sua azione: nella prima fase lancia il dado, nella seconda fase aggiorna la sua posizione, gestisce un eventualmente casella penalizzante e passa il turno.

Nel Risiko, invece, la turnazione è gestita con sei diverse fasi in cui, a seconda dell'azione svolta, si passa o meno il turno al giocatore avversario. Come si può immaginare tale sistema risulta quantomeno poco conciso, oltre che poco semplice.

Nel caso di Dungeon Crawl, la turnazione è più particolare, nel senso che non si tratta di passare il turno agli avversari ma ai “compagni di gioco”. In ogni modo, questo aspetto del gioco è gestito in maniera simile a quella del gioco del Risiko: a seconda dell'azione svolta, dei risultati degli scontri tra giocatori e mostri, e dei punti ferita dei giocatori, si passa o meno il turno al giocatore successivo.

Nel gioco Lotta a Squadre, invece, è presente un modulo turni che gestisce, con un sistema di etichette, il passaggio di turno da una squadra a un'altra, e all'interno del modulo delle squadre, una variabile che passa dal valore zero al valore due per il passaggio di testimone da un giocatore a un altro della stessa squadra.

In tutti questi casi differenti si può ipotizzare, con un linguaggio ad hoc, di gestire la turnazione implementando delle variabili che di volta in volta memorizzano chi è il giocatore o la squadra corrente, e l'aggiornamento di tale variabile ogni qualvolta è necessario passare il turno a un altro giocatore o a un'altra squadra.

6 DEFINIZIONE E DESCRIZIONE DEL NUOVO LINGUAGGIO

Il nuovo linguaggio creato nel lavoro di sperimentazione da me svolto ha sostanzialmente l'aspetto di un linguaggio di programmazione ad oggetti.

La modellazione fatta con il linguaggio di PRISM potrebbe essere vista come una ulteriore semplificazione della programmazione imperativa. La programmazione imperativa è un paradigma di programmazione in cui vengono utilizzati i seguenti tipi di dati:

- tipi di dato primitivi (integer, double, boolean, e così via);
- stringhe;
- array.

I programmi consistono in una sequenza di comandi, strutture di controllo e, eventualmente, metodi ausiliari e prevedono uno stato globale dato dal valore delle sue variabili.

La modellazione fatta con il linguaggio di PRISM può essere vista come una semplificazione di quest'ultima dal momento che prevede l'utilizzo di soli variabili e comandi; anche nella modellazione svolta con PRISM, il sistema ha diversi stati dati dal valore che le variabili assumono durante l'azione.

Sebbene sia possibile scrivere programmi interessanti anche con un tipo di programmazione imperativa, spesso si ha bisogno di manipolare strutture dati che rappresentano più fedelmente le entità del mondo reale: basti pensare ai conti bancari, o a un programma per la gestione dei dipendenti di un'azienda, ma senza andare molto lontano, basta pensare ai giochi.

Per questo motivo un linguaggio di programmazione orientato agli oggetti fornisce meccanismi per definire nuovi tipi di dato basati sul concetto di classe.

La programmazione orientata agli oggetti (OOP, *Object Oriented Programming*) è,

appunto, un paradigma di programmazione che permette di definire oggetti in grado di interagire gli uni con gli altri. Così come ogni entità del mondo reale prevede un proprio stato interno e delle proprie funzionalità, una classe definisce un insieme di oggetti dotati di proprie variabili, che rappresentano il suo stato interno, e di propri metodi, che realizzano le sue funzionalità.

Riportando il discorso sulla modellazione dei giochi: è vero che con PRISM model checker e il suo linguaggio è possibile costruire modelli abbastanza fedeli ai sistemi reali, come si può notare dai modelli implementati (consultabili nell'Appendice B), ma è anche vero che risulta complicato e poco conciso.

L'ideazione e la creazione di un linguaggio, dunque, che ha l'aspetto di un linguaggio di programmazione a oggetti e che può, quindi, gestire i vari elementi dei giochi come oggetti con i propri metodi e le proprie funzionalità, appare come una soluzione ideale.

Tuttavia, poiché come è stato descritto nel capitolo precedente, PRISM model checker presenta numerosi vantaggi, soprattutto per quel che riguarda la verifica delle proprietà , si è pensato di creare un linguaggio che, pur essendo di alto livello, potesse essere tradotto agevolmente nel linguaggio di PRISM in un futuro sviluppo del lavoro di sperimentazione da me svolto.

Il nuovo linguaggio di modellazione è stato chiamato GameDSL.

6.1 Descrizione di GameDSL

Come detto poco sopra, il nuovo linguaggio ha strutture dati simili a quelle di Java; queste ultime, però, sono state semplificate, in modo che in una futura traduzione in PRISM non si riscontrino eccessivi problemi.

Per fare questo è stato creato un Domain Specific Language che permette di trascrivere i modelli di gioco in poche righe di codice. Tale caratteristica si può notare anche solo guardando gli allegati, dai quali si può constatare la differente lunghezza dei modelli implementati in PRISM, nell'Appendice B, e quelli implementati in GameDSL, nell'Appendice A.

Il nuovo linguaggio implementato presenta strutture primitive, come liste e grafi, classi e oggetti, istruzioni condizionali del tipo if-then-else e un singolo ciclo globale ottenuto con termination e step.

In particolare i programmi GameDSL sono strutturati nella seguente maniera:

- Tutte le azioni del gioco sono inserite all'interno di un ciclo globale che, come vedremo più avanti, ha l'aspetto di un ciclo while: ogni singola azione avviene all'interno di uno step e il loro susseguirsi termina quando la condizione di terminazione risulta vera.

- Tutti gli elementi del gioco sono inseriti in una struttura dati di tipo lista e in alcuni casi in una struttura dati di tipo grafo. Le liste e i grafi sono degli elementi con struttura fissa: ciò significa che il numero degli elementi contenuti in tali strutture è fissato al momento della loro creazione e rimane tale, non può essere in alcun modo modificato.

- Le operazioni che si possono compiere su liste e grafi, che verranno descritti in seguito, non li modificano ma ne creano degli altri i cui elementi sono riferimenti agli elementi creati inizialmente.

- Gli elementi fondamentali del gioco sono stati implementati come classi.

Vediamo questi costrutti più nel dettaglio.

6.1.1 Tipi di dato

Oltre i tipi di dati primitivi (integer, double e booleani) e i dati di tipo classe, che saranno illustrati più avanti, è stato creato un tipo di dato chiamato `list`: l'idea è quella di mettere in un'unica struttura tutti gli oggetti simili.

La lista ha una struttura fissa: il numero di elementi presenti al suo interno viene definito al momento della sua creazione e tale rimane.

Questo aspetto risulta ottimale in vista di una futura traduzione di GameDSL nel linguaggio di PRISM: in PRISM, infatti, una volta create le variabili non possono essere eliminate. La struttura fissa delle liste permette di determinare in fase di compilazione il numero di variabili necessarie per tradurle.

Sulle liste è possibile, però, fare determinate operazioni:

- utilizzando il metodo `next()`: data una lista, restituisce l'elemento della lista

successivo a quello posto come parametro. Il seguente esempio mostra l'aggiornamento della variabile `current_player`: ipotizzando di avere una lista `P` di giocatori e ipotizzando che la variabile `current_player` memorizzi il riferimento al primo elemento della lista, con tale aggiornamento la variabile memorizza il riferimento al secondo giocatore della lista.

```
current_player = P.next(current_player);
```

- utilizzando il comando `filter`: data una lista, la filtra secondo una data condizione posta tra graffe. Il seguente esempio mostra in che modo una lista può essere filtrata: ipotizzando di avere un lista di giocatori `G`, il comando `filter` filtra la lista e ne crea un'altra con i riferimenti a tutti i `Giocatori g` di `G` i cui punti sono maggiori di zero.

```
filter(Giocatore g, G){g.punti>0};
```

Il comando `filter` in una futura traduzione del linguaggio GameDSL dovrebbe essere realizzato utilizzando le formule di PRISM (mostrate nella sezione 5.1). Bisognerà però tenere conto che la lista ha una struttura fissa, dunque il numero dei suoi elementi rimane invariato, e che, quando si crea una seconda lista tramite tale comando, gli elementi della seconda lista sono riferimenti agli oggetti della prima e non essi stessi degli oggetti nuovi.

- utilizzando le espressioni `exist` e `forall`:

- `exist`: data una lista di oggetti restituisce true se in tale lista esiste almeno un elemento che soddisfa una determinata condizione. Il seguente esempio mostra il funzionamento dell'espressione `exists`: ipotizzando di avere una lista `G` di `Giocatori g`, l'espressione restituisce `true` se la variabile casella di almeno un dei `Giocatori g` ha valore pari a 63; `false` altrimenti.

```
exists (Giocatore g, G) {g.casella==63};
```

- `forall`: data una lista di oggetti restituisce true se in tale lista tutti gli elementi soddisfano una determinata condizione. Il seguente esempio mostra il funzionamento dell'espressione `forall`: ipotizzando di avere una lista `A` di `Territori t`, l'espressione restituisce `true` se le variabili `propr` di ogni territorio t sono uguali alla variabile `propr` del primo elemento della lista A di territori; `false` altrimenti.

```
forall (Territorio t, A) {t.propr==A[0].propr};
```

L'altro tipo di dato creato, molto simile alla lista, è un tipo di dato grafo, inizializzato con la parola chiave `graph`. Anche il grafo ha, come la lista, una struttura fissa: il numero degli elementi presenti al suo interno viene definito al momento della sua creazione e tale rimane fino alla fine. Anche in questo caso questa caratteristica risulta ottimale per una futura traduzione di GameDSL nel linguaggio di PRISM, per lo stesso motivo espresso precedentemente per le liste.

Questo tipo di dato è stato usato nel gioco del Risiko, dove era necessario trovare un metodo per descrivere adeguatamente la plancia di gioco, formata da sei territori connessi tra loro da otto collegamenti. I collegamenti sono espressi tramite una lista particolare creata con la parola chiave `edges`.

```
edges E = {(0,1),(0,2),(1,3),(1,4),(2,3),(2,4),(3,5),(4,5)}
```

ogni elemento corrisponde a un link, nell'esempio riportato ad esempio l'elemento (0, 1) corrisponde al collegamento presente tra l'elemento che nella lista di territori ha indice zero e l'elemento che nella lista dei territori ha indice uno. Gli elementi sono riferimenti ai territori presenti nella lista dei territori.

A partire dalla lista dei territori e dai link viene creato il grafo

```
graph <Territorio> G = {A, E}
```

L'unica differenza tra il grafo e la lista sta nella dichiarazione del tipo di elementi che formano il grafo, indicato tra parentesi angolate.

Il grafo può essere modificato dal metodo `adj`, che restituisce la lista dei nodi adiacenti al nodo passato come parametro:

`G.adj(t)` ad esempio, restituisce i territori adiacenti al territorio t.

6.1.2 Ciclo globale

Uno degli aspetti che possiamo definire limitativo nel linguaggio di PRISM è il fatto che le iterazioni si realizzano con transizioni cicliche, ovvero che riportano allo stesso stato. A fronte del fatto che le attività ludiche sono spesso caratterizzate dalla ripetizione

di azioni simili, se non uguali – basti pensare al susseguirsi dei turni – ma che devono portare a uno stato diverso dal precedente, questa limitazione risulta costrittiva.

È stato, perciò, pensato di implementare un singolo ciclo globale ottenuto con le parole chiave `step` e `termination`.

La forma in cui il ciclo globale, con lo step e la condizione di terminazione, è scritto è molto simile a un ciclo while:

```
termination { espressione }
step {
corpo
}
```

I modelli dei giochi in sostanza sono suddivisi in piccoli passi che si susseguono fino a che la condizione di terminazione non risulta vera. Per utilizzare lo step, ogni modello definisce un `current_player` e, a seconda dei casi, anche altri elementi correnti (`current_team, current_room, current_monster` e via dicendo). In particolare:

1. Nel Gioco dell'Oca il singolo passo è rappresentato dal lancio del dado di uno dei giocatori e dall'aggiornamento della sua posizione nel tabellone e la condizione di terminazione è rappresentata dall'arrivo alla casella finale.

2. Nel Risiko il giocatore corrente può:

   - Non attaccare e passare il turno (aggiornamento della variabile `current_player`);

   - Attaccare definendo, tramite metodi opportuni, quale territorio attaccare e da quale territorio attaccare; in questo caso si procede con l'attacco e con l'eventuale aggiornamento del numero delle armate.

   La condizione di terminazione è che uno dei due giocatori possegga tutti i territori.

3. In Dungeon Crawl:

   - Si valuta se la `current_room` è la stanza finale o meno: in caso positivo si procede all'attacco previsto per la stanza finale;

- Se la stanza non è quella finale si procede all'attacco: in caso di sconfitta del mostro, si valuta se era l'ultimo mostro presente e in quel caso si aggiorna la variabile `current_room`; se non era l'ultimo mostro si aggiorna la variabile `current_monster`; se l'attacco è stato vinto dal mostro si aggiorna la variabile che memorizza i punti ferita del giocatore.

La condizione di terminazione è che abbiano vinto i giocatori, ovvero che siano arrivati alla stanza finale e abbiano sconfitto il mostro finale; oppure che abbiano vinto i mostri, ovvero che tutti i giocatori siano morti.

4. In Lotta a Squadre si valuta se il giocatore è vivo e in caso positivo si procede all'attacco; si valuta se il giocatore è l'ultimo della squadra corrente (`current_team`) e in caso positivo si aggiorna la variabile `current_team` e la variabile `current_player`, in caso negativo si aggiorna solamente la variabile `current_player`.
   La condizione di terminazione è che tutti i giocatori di una delle due squadre siano tutti morti.

6.1.3 Costrutto if-else

Un altro dei costrutti ripresi da Java sono le istruzioni condizionali if-else, semplici da tradurre in una futura traduzione di GameDSL nel linguaggio di PRISM.

Di fatto i comandi presenti nel linguaggio di PRISM sono molto simili al costrutto if-else: se la guardia risulta vera si aggiornano le variabili, altrimenti si tralascia il comando.

Il comando PRISM

```
[] phase = 2 & can_attack_count = 0 ->
                  (phase' = 0) & (current_player' = other_player);
```

potrebbe essere facilmente riscritto con un costrutto if-else, in questo modo:

```
if(phase = 2 & can_attack_count = 0){
   (phase = 0) & (current_player = other_player);
}
```

6.1.4 Le classi e il costruttore init

Gli elementi fondamentali del gioco sono stati implementati come classi: ad esempio la classe Giocatore è stata definita in ogni modello. Ogni classe ha al suo interno delle variabili che lo definiscono (ad esempio i punti ferita o la posizione nel tabellone) e, i diversi metodi che sono necessari per giocare.

La classe Giocatore nel modello del Gioco dell'Oca, per esempio, è così definita:

```
class Giocatore{
   int casella;
   int procedi(){
      casella = casella + random(1,6);
      if(casella>63){
         casella = 63 – (casella -63);
      }
   }
}
```

Le classi possono avere o meno un costruttore, dichiarato con la parola chiave `init`. La presenza o meno del costruttore condiziona il metodo di inizializzazione dell'oggetto: se è presente, quando si inizializza un oggetto, tra parentesi si mettono i parametri del costruttore; se non è presente, quando si inizializza l'oggetto, tra parentesi si mettono i valori delle variabili della classe che non sono già inizializzate nell'ordine giusto. Di seguito vengono mostrati due esempi dei due diversi casi.

Nel primo caso è presente il costruttore: al momento dell'inizializzazione della lista di oggetti Stanza, tra parentesi come parametro viene posto `int n`, che in questo caso è un intero generato in maniera random nell'intervallo (4, 7):

```
class Stanza {
   list Mostro M = {(1), (1), (1), (1), (1), (1), (1)};
   int num_mostri;
   init(int n) {
      num_mostri = n;
   }
}

list Stanza A = {(random(4,7)), (random(4,7)), (random(4,7))}
```

Nel secondo caso il costruttore non è presente: al momento dell'inizializzazione della lista di oggetti Giocatore, tra parentesi viene messo il valore della prima e unica variabile presente nella classe Giocatore, ovvero l'intero `puntiFerita`.

```
class Giocatore {
   int puntiFerita;
   int lanciaDado(){
      return random(1,6);
   }
}

list Giocatori G = {(7), (7), (7), (7)};
```

La parola chiave `init` viene usata anche per l'inizializzazione delle variabili globali: ad esempio, nel modello del Risiko, viene utilizzato per posizionare i rinforzi all'inizio della partita.

```
init{

current_player.posizionaArmate(filter(Territorio t, A)
{t.proprietario==current_player}).size/2);

}
```

6.1.5 Comandi per le scelte probabilistiche

Ogni modello può usare, a seconda delle esigenze, dei comandi per le scelte probabilistiche:

1. `distribution`: restituisce una distribuzione di una probabilità da usare per scegliere un elemento da una lista. Il seguente esempio mostra la definizione di una distribuzione d che attribuisce la stessa probabilità ad ogni elemento t della lista L di oggetti di tipo Territorio: il seguente tipo di distribuzione è detto distribuzione uniforme.

   ```
   distribution d(Territorio t, L) { 1/L.size }
   ```

2. `pick`: seleziona casualmente un elemento di una lista secondo la distribuzione

indicata. Il seguente esempio seleziona in maniera casuale un elemento della lista G di giocatore secondo la distribuzione d.

```
pick(G, d);
```

3. `random`: restituisce un intero random compreso nell'intervallo espresso tra i parametri. Il seguente esempio genera un numero casuale compreso tra uno e sei.

```
Random(1,6);
```

I comandi per le scelte deterministiche potranno essere realizzati tramite delle formule in una futura traduzione di GameDSL nel linguaggio di PRISM. Ad esempio nella scelta del territorio da attaccare, come già detto, nel modello PRISM, sono presenti:

- sei formule per indicare se il giocatore corrente possiede almeno due armate in un territorio;
- due formule per indicare il numero di territori posseduti da ciascun giocatore;
- sei formule per indicare se il giocatore corrente possiede o meno un territorio;
- sei formule per indicare se l'avversario del giocatore corrente possiede un territorio;
- sei formule per indicare se il giocatore corrente può attaccare un territorio
- una formula per indicare il numero dei territori dal quale è possibile lanciare un attacco;

In base al risultato di tutte queste formule si esegue un comando che determina da quale territorio attaccare.

In GameDSL tutte queste formule sarebbero la traduzione del comando:

```
Territorio ScegliChiAttacca(Territorio dif){
   return pick(filter(Territorio t, G.adj(dif))
{t.proprietario==this && t.armate>1},d);
}
```

in cui si filtra la lista dei territori adiacenti all'elemento passato come parametro, ovvero il territorio che si è deciso di attaccare, secondo la condizione che sia di proprietà del giocatore corrente e possegga almeno un'armata, e da questa lista viene selezionato casualmente un territorio secondo la distribuzione d.

6.1.6 I metodi

Oltre ai metodi per le strutture dati già visti sono stati definiti metodi specifici a seconda delle dinamiche di gioco: troviamo ad esempio il metodo `lanciaDado()` (presente in quasi tutti i modelli di gioco), il metodo `scegliChiAttaccare()`, il metodo `attacca()` e via dicendo.

I metodi sono scritti in maniera molto simile a come sono scritti in Java. L'unica differenza è che non è presente un modificatore ma solo il tipo del risultato:

```
boolean attacca(){
   int random = random(0, 100);
      if(random>50 & (filter(Territorio t, A){t.proprietario!=this
&& filter(Territorio t2, G.adj(t)){t2.proprietario==this &&
t2.armate>2}.size>0){
         return true;
      }else{
         return false;
      }
}
```

Nel presente esempio si genera casualmente un intero tra zero e cento; dopo di ciò, posto che i territori t, filtrati dalla lista dei territori, appartengano all'avversario del giocatore corrente e che esistano territori adiacenti ad essi di proprietà del giocatore corrente e posto che il comando random restituisca un valore maggiore di 50, il metodo `attacca()` restituisce il valore `true; false` altrimenti.

Per tutto il resto è praticamente uguale a un metodo del linguaggio Java, con l'uso della parola chiave `return` per la restituzione del risultato e la lista dei parametri (eventualmente vuota) tra parentesi.

6.2 Grammatica di GameDSL

Come detto nel primo Capitolo, le grammatiche presentano diversi formalismi attraverso i quali è possibile esprimere le regole di derivazione. La grammatica del nuovo linguaggio è stata definita attraverso la metasintassi *Backus-Naur Form* (BNF).

Di seguito le regole grammaticali del nuovo linguaggio e la sua descrizione.

Come già detto una grammatica G è una quadrupla $G = \{N, \Sigma, S, P\}$, dove N è un insieme finito di simboli che rappresentano categorie sintattiche o simboli non terminali, Σ è l'alfabeto, S è un simbolo di categoria sintattica indicato come iniziale o principale, appartenente a N e P è un insieme di regole di produzione.

Nell'insieme dei non terminali N del nuovo linguaggio troviamo tutte quelle categorie sintattiche che vogliamo definire, in particolar modo:

$N = \{Model, ClassDef, VarDef, BasicType, CompoundType, DistDef, MethodDef, Tuple, Expr, Com\}$

Vediamole nel dettaglio.

Un modello è:

- una sequenza anche vuota di classi (ClassDef) , di variabili (VarDef) o di distribuzioni (DistDef)
- eventualmente la parola chiave init seguita da un comando racchiuso tra graffe,
- la parola chiave termination seguita da un espressione racchiusa tra graffe,
- la parola chiave step seguita da una sequenza, eventualmente vuota, di comandi racchiusi tra graffe.

```
Model ::= (ClassDef | VarDef | DistDef)* ['init' '{' Com '}']
'termination''{' Expr '}' 'step' '{' Com* '}'
```

Una definizione di variabile è:

- un tipo base (BasicType) seguito da un ID ed eventualmente un uguale e un'espressione (Expr) , il tutto seguito da un punto e virgola:

  ```
  int x = 5;
  ```

  ```
  int x;
  ```

- oppure la parola chiave list seguita da un tipo base e un ID, un uguale e eventualmente una lista, anche vuota, di tuple seguite da una virgola, racchiusa tra graffe, il tutto seguito da un punto e virgola

  ```
  list Giocatori G = {(7), (7), (7), (7)};
  ```

  ```
  list Territori T = {};
  ```

- oppure la parola chiave graph seguita da un tipo base e un ID, un uguale e tra graffe eventualmente una lista di uno o più ID

- oppure la parola chiave edges seguita da un ID e un uguale e tra parentesi graffe una o più coppie di interi tra tonde

```
 VarDef ::= (BasicType ID ['=' Expr] |
'list' BasicType ID '=' '{' [(Tuple ',')* Tuple]'}') ';'|
'graph' BasicType ID '=' '{'[(ID',')*ID]'}' ';' |
'edges' ID '=' '{'('('<INTEGER_LITERAL> ',' <INTEGER_LITERAL> ')'
',')*'('<INTEGER_LITERAL> ',' <INTEGER_LITERAL> ')' '}' ';'
```

Un tipo base può essere:

- un intero
- un booleano
- un double

- un ID

  ```
  list Giocatore G = {};
  ```

  in questo caso Giocatore è un tipo base.

```
BasicType ::= 'int' | 'bool' | 'double' | ID
```

Un tipo composto può essere:

- un tipo base
- la parola chiave list seguita da un tipo base

```
CompoundType ::= 'list' BasicType | BasicType
```

Le tuple sono un'eventuale lista di espressioni e virgole racchiusa tra parentesi tonde.

```
Tuple ::= '(' [(Expr ',')* Expr] ')'
```

Una classe è:

- la parola chiave class seguita da un ID (un nome identificativo scelto da chi implementa la classe) seguita da una lista, anche vuota, di variabili (VarDef) o di metodi (MethodDef), racchiusa tra graffe.

```
ClassDef ::= 'class' ID '{' (VarDef | MethodDef)* '}'
```

Una distribuzione è:

- la parola chiave distribution seguita da un ID seguita da un tipo base con un ID e un ID, divisi da una virgola e tra tonde, il tutto seguito da un'espressione tra graffe.

```
DistDef ::= 'distribution' ID '('BasicType ID ',' ID')''{'Expr'}'
```

Una definizione di metodo è: o un tipo composto o la parola chiave void seguito da un ID e delle parentesi tonde al cui interno si trova un tipo base con un ID o una lista di tipi base con un ID, seguiti da una lista, anche vuota, di comandi tra graffe.

```
MethodDef ::= (CompoundType | 'void') ID '('[BasicType ID',')*
BasicType ID]')' '{' Com* '}'
```

Un'espressione può essere sia:

- delle cifre numeriche (<INTEGER_LITERAL>)
- un ID
- due ID con un punto nel mezzo e eventualmente un espressione o una lista di esse
- un'espressione più un'altra espressione
- un'espressione meno un'altra espressione

- un'espressione per un'altra espressione
- un'espressione diviso un'altra espressione
- la parola chiave random con delle coppie di interi tra parentesi
- la parola chiave pick seguita da un'espressione una virgola e un ID tra parentesi tonde
- Un'espressione booleana seguita da un punto interrogativo, un'espressione, due punti e un'altra espressione
- la parola chiave filter con tra parentesi tonde un tipo base e un id e un id, seguiti da un'espressione booleana tra graffe
- il punto esclamativo seguito da un'espressione
- il segno della sottrazione seguito da un'espressione

che un'espressione booleana – sarà poi compito di una successiva fase di compilazione effettuare il type checking per valutare di che tipo sono le espressioni:

- un'espressione seguito dal simbolo di maggiore, il singolo di minore o il simbolo di uguaglianza, seguiti da un'espressione;
- due espressioni booleane con nel mezzo un and o un or
- le parole chiavi true o false
- la parola chiave forall seguita tra parentesi da un tipo base con un ID e un altro ID divisi da una virgola, seguiti da un'espressione booleana tra parentesi
- la parola chiave exists seguita tra parentesi da un tipo base con un ID e un altro ID divisi da una virgola, seguiti da un'espressione booleana tra parentesi

```
Expr ::= <INTEGER_LITERAL> | ID |
ID'.'ID ['('[(Expr',')* Expr]')'] | Expr '+' Expr |
Expr '-' Expr |  Expr '*' Expr |  Expr '/' Expr |
'random' '('<INTEGER_LITERAL>',' <INTEGER_LITERAL>')' |
'pick' '('Expr ',' ID) |Expr'?' Expr ':' Expr
'filter' '('BasicType' ID ',' ID ')''{'Expr'}'
| '!' Expr | '-' Expr |
Expr ('>' | '<' | '==') Expr | Expr ('&' | '|')Expr
| 'true' | 'false'|
'forall' '(' BasicType ID ',' ID ')' '{'Expr'}'';' |
'exists' '(' BasicType ID ',' ID ')' '{'Expr'}'';'
```

Un comando può essere:

- eventualmente un tipo composto seguito da un ID seguito da un uguale e un espressione, il tutto seguito da un punto e virgola
- la parola chiave if seguita da un'espressione booleana tra parentesi e una lista, anche vuota, di comandi tra graffe; eventualmente seguito dalla parola chiave else e una lista, anche vuota, di comandi tra graffe
- la parola chiave return seguita eventualmente da un'espressione
- due ID tra un punto seguiti da delle parentesi tonde con eventualmente un espressione dentro

```
Com ::= [CompoundType] ID '=' Expr ';' | 'if' '(' BEexpr ')'
'{' Com*'}' ['else' '{' Com*'}'] | 'return' [Expr]';' |
ID'.'ID '('[Expr]')'
```

7 ESEMPI DI GIOCHI RE-IMPLEMENTATI NEL NUOVO LINGUAGGIO E CONFRONTO CON I MODELLI IMPLEMENTATI IN PRISM

Come più volte espresso, i quattro casi studio sono stati re-implementati in maniera significativamente più semplice e concisa. Vediamoli nel dettaglio (per una visione di insieme nelle Appendici A e B sono presenti tutti i codici sorgente), analizzando le principali differenze tra i modelli implementati nel linguaggio di PRISM – che da qui in avanti chiameremo modelli PRISM – e i modelli implementati nel nuovo linguaggio – che da qui in avanti chiameremo modelli GameDSL.

Si specifica preliminarmente che quando si parla di "problemi" riscontrati con il linguaggio di PRISM, si intendono difficoltà dovute alla semplicità di tale linguaggio, e non di problematiche vere e proprie.

7.1 Il Gioco dell'Oca

Il modello del Gioco dell'Oca, in effetti, non presenta grossi problemi anche con l'implementazione in PRISM: risulta abbastanza semplice e conciso, o almeno lo è in confronto agli altri modelli.

Con il nuovo linguaggio, però, risulta enormemente semplificato: basti pensare che, se solo si volesse osservare l'aspetto quantitativo, il modello GameDSL consiste in meno di venti righe di codice.

Le principali differenze sono due: la prima riguarda la gestione dei turni, la seconda l'avanzamento sul tabellone.

Mentre nel modello PRISM la gestione dei turni è coordinata dal modulo Mturns:

```
module MTurns
next_step : [1..2] init initial_player;
[turn1] !terminated & next_step = 1 -> (next_step' = 2);
[turn2] !terminated & next_step = 2 -> (next_step' = 1);
endmodule
```

Nel modello GameDSL è gestita con il semplice aggiornamento della variabile `current_player`, grazie al metodo `next`:

```
current_player=G.next(current_player);
```

Per quel che riguarda l'avanzamento sul tabellone, nel modello PRISM troviamo un comando con sei diversi aggiornamenti con 1/6 di probabilità l'uno:

```
[turn1] !terminated ->
                    1/6 : (x' = (x + 1 <= cell_count) ? (x + 1) :
(2 * cell_count - x - 1)) +
                    1/6 : (x' = (x + 2 <= cell_count) ? (x + 2) :
(2 * cell_count - x - 2)) +
                    1/6 : (x' = (x + 3 <= cell_count) ? (x + 3) :
(2 * cell_count - x - 3)) +
                    1/6 : (x' = (x + 4 <= cell_count) ? (x + 4) :
(2 * cell_count - x - 4)) +
                    1/6 : (x' = (x + 5 <= cell_count) ? (x + 5) :
(2 * cell_count - x - 5)) +
                    1/6 : (x' = (x + 6 <= cell_count) ? (x + 6) :
(2 * cell_count - x - 6));
```

mentre nel modello GameDSL viene tutto svolto all'interno del metodo procedi:

```
int procedi(){
   casella=casella + random(1,6);
   if(casella> 63){
      casella=63-(casella-63);
   }
}
```

7.2 Risiko

Ciò che ancora con il modello del Gioco dell'Oca non si notava, risulta molto più chiaro con il modello del Risiko, ovvero ritornando sul semplice aspetto quantitativo, basta osservare i codici sorgente dei due modelli nelle sezioni B.2 (PRISM) e A.2

(GameDSL) – scritti peraltro con lo stesso carattere e la stessa dimensione – per notare come a fronte delle quasi dieci pagine di codice del modello PRISM, troviamo un modello GameDSL di poco più di due pagine.

Il principale problema presente nel modello PRISM del gioco del Risiko riguardava la definizione dei comandi per scegliere chi attaccare, scegliere da dove attaccare e per posizionare i rinforzi. Come avevo già accennato nel capitolo cinque, nella descrizione degli svantaggi del linguaggio di PRISM, queste azioni nel suddetto linguaggio risultano molto lunghe e complicate: rivediamole nel dettaglio facendo un confronto con l'implementazione nel modello GameDSL.

Come già detto in precedenza il modello presenta sei diverse variabili che memorizzano chi possiede il territorio (s1, s2, s3, s4, s5, s6) e sei diverse variabili che memorizzano il numero di armate possedute da ciascun territorio (c1, c2, c3, c4, c5, c6).

Di seguito verranno mostrate le formule e i comandi nel linguaggio di PRISM, atti a scegliere chi attaccare e da dove farlo. Si tenga conto che, per una questione di lunghezza, in alcuni casi riprenderò solo le formule per il territorio uno, anche se in realtà nel modello ogni formula è ripetuta sei volte: una per ogni territorio presente sulla plancia di gioco.

- Formula per indicare se il giocatore corrente possiede almeno due armate in un territorio (sono sei formule, una per ogni territorio):

```
formula cs1 = (s1 = current_player & c1 >= 2) ? 1 : 0;
```

- Formule che indicano il numero di territori posseduti da ciascun giocatore:

```
formula p1_territories_count = (1 - s1) + (1 - s2) + (1 - s3)
+ (1 - s4) + (1 - s5) + (1 – s6);

formula p2_territories_count = s1 + s2 + s3 + s4 + s5 + s6;
```

- formula che indica se il giocatore corrente possiede un territorio (sei formule, una ogni territorio):

```
formula current_player_owns_s1 = (current_player = player1 ?
(1 - s1) : s1);
```

- formula per inserire il rinforzo a ciascun territorio posseduto dal giocatore corrente (sei formule, una ogni territorio):

```
formula reinforcements_s1 =

current_player_owns_s1 / territories_count;
```

- formula che indica se l'altro giocatore possiede un terriorio (sei formule, una ogni territorio):

```
formula other_player_owns_s1 = 1 – current_player_owns_s1;
```

- formula che indica se il giocatore corrente può attaccare un territorio (sei formule, una ogni territorio):

```
formula can_attack_s1 = other_player_owns_s1 * max(cs2, cs3);
```

  in questo caso l'espressione max(cs2, cs3) varia a seconda dei territori che si intende attaccare.

- Formula che restituisce il numero dei territori dal quale è possibile lanciare un attacco da parte del giocatore corrente:

```
formula can_attack_count =
can_attack_s1 + can_attack_s2 + can_attack_s3 + can_attack_s4
+ can_attack_s5 + can_attack_s6;
```

- Comando che determina quale territorio attaccare:

```
[] phase = 2 & can_attack_count > 0 ->
                          can_attack_s1/can_attack_count :
(dif' = 1) & (phase' = 3) +
                          can_attack_s2/can_attack_count :
(dif' = 2) & (phase' = 3) +
                          can_attack_s3/can_attack_count :
(dif' = 3) & (phase' = 3) +
                          can_attack_s4/can_attack_count :
(dif' = 4) & (phase' = 3) +
                          can_attack_s5/can_attack_count :
```

```
(dif' = 5) & (phase' = 3) +
                          can_attack_s6/can_attack_count :
(dif' = 6) & (phase' = 3);
```

- Comando che determina da dove attaccare (sono in realtà cinque comandi differenti, a seconda di che valore abbia preso la variabile dif):

```
[] phase = 3  & dif = 1 & cs2 + cs3 > 0 ->
           cs2/(cs2 + cs3) : (att' = 2) & (phase' = 4) +
           cs3/(cs2 + cs3) : (att' = 3) & (phase' = 4);
```

- Comando che modella i possibili risultati di un attacco (sono in realtà sedici distinti comandi per ciascuna coppia att/dif):

```
[] phase = 4 & dif = 1 & att = 2 & !some_empty_territory ->
0.417  :  (c1'  =  c1  -  1)  &  (phase'  =  5)  +
0.583 : (c2' = c2 - 1) & (phase' = 5);
```

Tutte queste righe di codice sono espresse nel modello GameDSL dai seguenti tre metodi, definiti all'interno della classe Giocatore:

- Rinforzi:

```
void posizionaArmate(int n) {
   pick(filter(Territorio t, A){t.proprietario=this},
d).armate += n;
}
```

- Scegliere chi attaccare:

```
Territorio scegliChiAttaccare(){
   return pick(filter(Territorio t, A){t.proprietario!=this
&& filter(Territorio t2, G.adj(t)){t2.proprietario==this &&
t2.armate>1}.size > 0},d);
}
```

- Scegliere da dove attaccare:

```
Territorio ScegliChiAttacca(Territorio dif){
   return pick(filter(Territorio t, G.adj(dif))
t.proprietario==this && t.armate>1},d);
}
```

Il parametro Territorio dif viene inizializzato all'interno dello step, con il risultato del metodo ScegliChiAttaccare.

7.3 Dungeon Crawl

Anche in questo caso, possiamo notare nelle sezioni B.3 e A.3 le differenze nella lunghezza dei codici sorgente.

Nel caso del gioco collaborativo, i principali problemi riscontrati in PRISM riguardano i metodi con cui viene creato un numero random di mostri diverso per ogni stanza e il metodo con cui vengono gestiti gli attacchi tra giocatori e mostri, nonché l'attacco tra giocatori e mostro finale.

Per quel che riguarda la creazione di un numero random di mostri in PRISM viene gestito con un comando, i cui aggiornamenti di variabili seguono delle probabilità: con ¼ di probabilità la variabile `monsters` viene inizializzata a 4, 5, 6 o 7.

```
[room_enter] room < 10 & monsters = -1 ->
                                        1/4: (monsters' = 4) +
                                        1/4: (monsters' = 5) +
                                        1/4: (monsters' = 6) +
                                        1/4: (monsters' = 7);
```

Nel modello GameDSL, invece, viene inizializzata una lista di sette mostri all'interno della classe Stanza. All'interno di quest'ultima, inoltre, è stato posto un costruttore che prende come parametro un intero n, che rappresenta appunto il numero dei mostri.

Al momento della creazione della lista di stanze, il parametro n prende valore random(4, 7):

```
list Stanza A = {(random(4,7)), (random(4,7)), (random(4,7)),
(random(4,7)), (random(4,7)), (random(4,7)), (random(4,7)),
(random(4,7)), (random(4,7))};
```

La lista dei mostri rimane per ogni stanza, di lunghezza fissata a 7, ma a seconda del risultato dell'espressione random, viene creato un numero diverso di mostri. Questo è stato pensato per una futura traduzione nel linguaggio di PRISM: in PRISM non è possibile eliminare o aggiungere delle variabili, quindi è necessario che le strutture dati che si vogliono ritradurre nel suo linguaggio, siano di lunghezza fissa e non modificabili.

Per quel che riguarda l'attacco invece il modello PRISM lo gestisce con i seguenti comandi:

- Nel caso delle prime nove stanze:

```
[] room < 10 & monsters > 0 & cp_health > 0 & choice = 0 →
          player_wins_prob :
            (monsters' = monsters - 1) & (choice' = -1) &
            (current_player' = next_player) +
          1 - player_wins_prob :
            (g1' = cp_lost_update_g1) &
            (g2' = cp_lost_update_g2) &
            (g3' = cp_lost_update_g3) &
            (g4' = cp_lost_update_g4) &
            (extra_health' = extra_health -
(cp_gets_extra_health ? 1 : 0)) &
            (choice' = -1) &
            (current_player' = next_player);
```

- nel caso della stanza finale:

```
[phase1] monster_sum <= 6*3 →
                        1/6: (monster_sum' = monster_sum + 1) +
                        1/6: (monster_sum' = monster_sum + 2) +
                        1/6: (monster_sum' = monster_sum + 3) +
                        1/6: (monster_sum' = monster_sum + 4) +
                        1/6: (monster_sum' = monster_sum + 5) +
                        1/6: (monster_sum' = monster_sum + 6);
```

```
[phase2] players_sum <= 6*3 ->
                        1/36: (players_sum' = players_sum + 1) +
                        3/36: (players_sum' = players_sum + 2) +
                        5/36: (players_sum' = players_sum + 3) +
                        7/36: (players_sum' = players_sum + 4) +
                        9/36: (players_sum' = players_sum + 5) +
                        11/36: (players_sum' = players_sum + 6);
```

Nel modello GameDSL, invece, tutti gli attacchi, sia nelle prime nove stanze che nella stanza finale sono gestiti in maniera molto semplice grazie al metodo `lancioDadi()`, implementato nella classe Giocatore, nella classe Mostro e nella classe StanzaFinale: in questo modo ogni qual volta un giocatore deve combattere contro un mostro o il mostro finale, è sufficiente richiamare il metodo sul giocatore corrente o sul mostro corrente; nel caso del mostro finale, il metodo `lanciaDadi()` è richiamato direttamente sull'oggetto stanza finale:

```
StanzaFinale sf = ();
   if (sf.lanciaDadi()>lancioFinale) {
      vinconoMostri=true;
   } else {
      vinconoGiocatori = true;
   }
}
```

7.4 Lotta a Squadre

Nel caso del modello del gioco Lotta a Squadre, ciò che era risultato problematico in PRISM era stato l'esprimere le due differenti strategie.

In particolar modo è stato necessario definire un grande numero di formule diverse:

- Tre formule (una per ciascun giocatore) che indicano i punti persi:

  ```
  formula puntiPersiA = 10 - a;
  formula puntiPersiB = 10 - b;
  formula puntiPersiC = 10 - c;
  ```

- Tre formule (una per ciascun giocatore) che serviranno per determinare le probabilità di attacco di ciascun giocatore nella strategia con le probabilità variabili a seconda dei punti persi:

```
formula attaccoA = (puntiPersiA = 10) ? 0 : puntiPersiA+5;
formula attaccoB = (puntiPersiB = 10) ? 0 : puntiPersiB+5;
formula attaccoC = (puntiPersiC = 10) ? 0 : puntiPersiC+5;
```

- Sei formule (per ogni giocatore di ciascuna squadra) che indicano se un giocatore è vivo o meno:

```
formula xVivo = (x>0)?1:0;
formula yVivo = (y>0)?1:0;
formula zVivo = (z>0)?1:0;
formula aVivo = (a>0)?1:0;
formula bVivo = (b>0)?1:0;
formula cVivo = (c>0)?1:0;
```

- Due formule (una per ciascuna squadra) che indicano il numero di giocatori vivi:

```
formula numVivi = xVivo + yVivo + zVivo;
formula numVivi2 = aVivo + bVivo + cVivo;
```

- Tre formule (una per ciascun giocatore) che indicano il giocatore con meno punti, che serviranno poi nella strategia in cui viene attaccato sempre il giocatore con meno punti:

```
formula PerdenteX= ((x<y & x<z & x>0) | (y=0 & x<z & x>0) |
(z=0 & x<y & x>0) | (y=0 & z=0 & x>0))? numVivi:0;

formula PerdenteY= ((y<x & y<z & y>0) | (x=0 & y<z & y>0) |
(z=0 & y<x & y>0) | (x=0 & z=0 & y>0))? numVivi:0;

formula PerdenteZ= ((z<x & z<y & z>0) | (x=0 & z<y & z>0) |
(y=0 & z<x & z>0) | (x=0 & y=0 & z>0))? numVivi:0;
```

- Due formule (una per ogni squadra) che indicano se i giocatori di una squadra hanno tutti lo stesso numero di punti ferita:

```
formula tuttiUguali1 = ((a=b & a=c & a>0) | (a=b & c=0) |
(a=c & b=0) | (b=c & a=0));

formula tuttiUguali2 = ((x=y & x=z & x>0) | (x=y & z=0) |
(x=z & y=0) | (y=z & x=0));
```

Oltre a queste formule gli attacchi sono gestiti con dei comandi. Nel caso della strategia in cui le probabilità di attacco variano a seconda dei punti persi il comando è il seguente:

```
[]!terminato & turnoSq1=0 & dado1!=0 & !tuttiUguali1->
                        attaccoA/(attaccoA+attaccoB+attaccoC)   :
(a'= max(a-dado1, 0)) & (turnoSq1'=1) & (dado1'=0)+
                        attaccoB/(attaccoA+attaccoB+attaccoC)  :
(b'= max(b-dado1, 0)) & (turnoSq1'=1) & (dado1'=0)+
                        attaccoC/(attaccoA+attaccoB+attaccoC)   :
(c'= max(c-dado1, 0)) & (turnoSq1'=1) & (dado1'=0);
```

Nel caso della strategia in cui viene attaccato sempre il giocatore con meno punti finché non muore, il comando è il seguente:

```
[]!terminato & turnoSq2=0 & dado2!=0 & !tuttiUguali2 & !anomalo2->
                        PerdenteX/numVivi : (x'= max(x-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0)+
                        PerdenteY/numVivi : (y'= max(y-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0)+
                        PerdenteZ/numVivi : (z'= max(z-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0);
```

8 IMPLEMENTAZIONE DEL PARSER

Affinché i programmi scritti in un dato linguaggio di programmazione *L* possano essere eseguiti da una macchina, è necessario che la suddetta macchina venga implementata. Per fare ciò è possibile utilizzare fondamentalmente tre tecniche [25]:

1. Realizzazione hardware: si realizzano in hardware tutte le componenti della macchina.

2. Realizzazione interpretativa: richiede l'esistenza di una macchina già realizzata, una macchina ospite. La tecnica consiste nel realizzare tutte le componenti della macchina che si vuole interpretare mediante programmi e strutture dati del linguaggio della macchina ospite.

3. Realizzazione compilativa: richiede, anche in questo caso, l'esistenza di una macchina già realizzata. La tecnica compilativa si basa sull'idea di tradurre l'intero programma scritto in *L* in un programma funzionalmente equivalente scritto nel linguaggio della macchina ospite. Il compito di eseguire tale traduzione è eseguito da un programma detto compilatore.

Tralasciamo la prima tecnica che risulta molto costoso sia dal punto di vista della progettazione che dal punto di vista della realizzazione.

Utilizzare la tecnica interpretativa significa in sostanza avere una macchina, chiamata appunto interprete, che prende il programma con il nuovo linguaggio *L* e lo esegue.

Utilizzare la tecnica compilativa, invece, significa avere una macchina, chiamata appunto compilatore, che prende un programma scritto nel linguaggio sorgente e restituisce un programma equivalente a quello dato, ma scritto in un linguaggio target, che è il linguaggio della macchina intermedia scelta per l'implementazione. Questo è quello che avviene quando, per esempio, si esegue su una macchina un programma Java: la macchina prende il programma e lo riscrive nel suo linguaggio, ovvero in bytecode.

Il compilatore, dunque, non è altro che una macchina che, preso un input *P*, restituisce un output *R*.

La conoscenza di tecniche efficaci per la scrittura di un programma compilatore è ad

oggi molto vasta e strutturata. Non era così per i primi programmatori che alla fine degli anni '50 si accingevano a scriverne uno. Si pensi ad esempio che la scrittura del primo compilatore Fortran richiese 18 anni di lavoro da parte di uno staff. Sin da allora sono state individuate molte tecniche sistematiche per la scrittura del codice delle fasi più importanti di un compilatore, oltre ad essere sorti linguaggi di programmazione adatti per l'implementazione, ambienti di programmazione e tool software, ovvero generatori automatici di codice a partire dalle specifiche di varie parti del linguaggio sorgente.

Il processo di compilazione è concettualmente diviso in due parti: l'analisi e la sintesi. La parte di analisi si occupa di dare una struttura al codice sorgente e di creare una sua rappresentazione intermedia. La parte di sintesi genera il codice nel linguaggio target a partire dalla rappresentazione intermedia [14].

Durante la fase di analisi le operazioni indicate nel programma sorgente vengono riconosciute e raggruppate in una struttura ad albero. Spesso viene usato il cosiddetto albero di sintassi astratta (*Abstract Sintax Tree*, AST).

8.1 Che cos'è un parser e come funziona

Come detto poco sopra, durante la fase di compilazione, il programma P – input dato alla macchina nel linguaggio di programmazione L – affronta una fase di analisi in cui le strutture del programma sorgente vengono esaminate. Questa parte di analisi è svolta, appunto, dal parser.

L'analisi viene divisa in tre fasi:

- Analisi lessicale;
- Analisi sintattica;
- Analisi semantica.

Durante la fase di analisi lessicale, detta anche analisi lineare o scanning, lo stream di caratteri che compongono il programma viene letto da sinistra a destra e raggruppato in cosiddetti token. I token sono sequenze di caratteri che hanno un significato comune.

Durante questa analisi una linea di caratteri come la seguente

```
position ::= initial + rate * 60
```

viene raggruppata nei seguenti token:

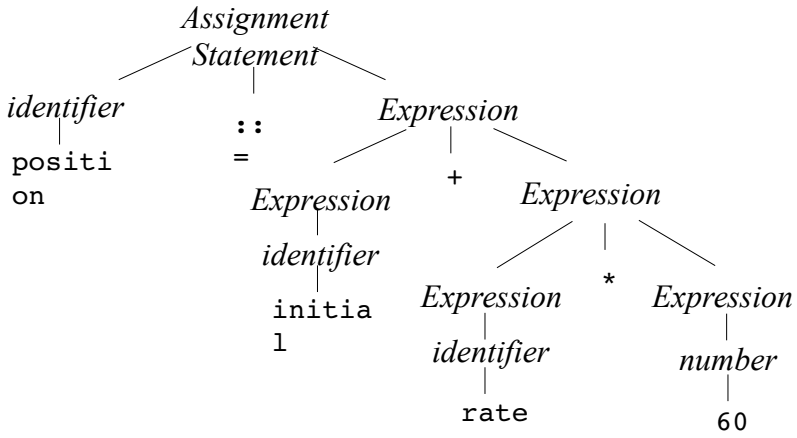
1. L'identificatore `position`
2. Il simbolo di assegnamento `::=`
3. L'identificatore `initial`
4. Il segno dell'addizione
5. L'identificatore `rate`
6. Il segno della moltiplicazione
7. Il numero 60

I caratteri di spazio che separano i token, le tabulazioni, i commenti e i caratteri che indicano la fine di una linea vengono normalmente eliminati durante la fase di analisi lessicale.

In sostanza, il compito della fase di analisi lessicale è quello di fornire alle fasi successive uno stream di token: l'analizzatore chiama un metodo o una funzione, a seconda del linguaggio in cui è costruito, e il parser ottiene un nuovo token; per fare ciò l'analizzatore legge i caratteri uno dopo l'altro fino al riconoscimento di un token. Nel caso non venga riconosciuto nessun token viene segnalato un errore.

Durante la fase di analisi sintattica, chiamata anche analisi gerarchica, i token vengono raggruppati in frasi grammaticali che poi saranno usate per sintetizzare l'output. Generalmente le frasi grammaticali del programma sorgente sono rappresentate con un albero di derivazione, anche detto *parse tree*, di una grammatica libera dal contesto.

Di seguito un esempio di albero di derivazione creato dalla sequenza di token usata come esempio precedentemente:



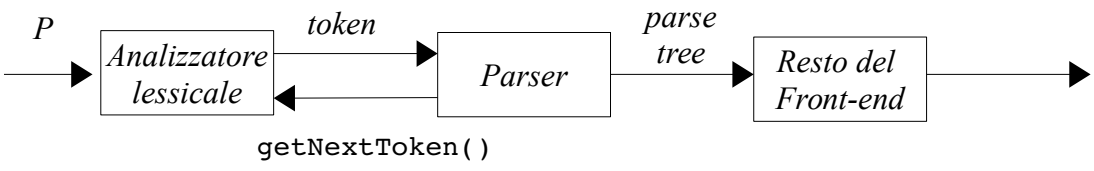
Supponendo di avere un linguaggio *L* con una grammatica *G(L)*, l'analisi sintattica è la verifica che il programma scritto con *L* e sottoposto a tale analisi, segua le regole grammaticali di *G(L)*.

La fase di analisi semantica controlla se sono presenti errori semantici nel programma sorgente e acquisisce l'informazione sui tipi che verrà usata nella fase successiva di generazione del codice. La rappresentazione gerarchica generata dall'analisi sintattica viene usata per identificare gli operatori e gli operandi delle espressioni e degli statement.

Una fase importante di questa fase di analisi è il *type checking* in cui il compilatore controlla che ogni operatore venga applicato ad operandi consentiti dalla specifica del linguaggio.

Dunque, il parser è lo strumento il quale, ottenuta una stringa di token dall'analizzatore lessicale, tenta di costruire l'albero di derivazione dalla stringa di token in accordo alle regole della grammatica del linguaggio.

Di seguito vengono mostrate sinteticamente le fasi del compilatore descritte poco sopra:



in cui:

- *P* è il programma di input scritto con il linguaggio *L*;

- `getNextToken()` è il metodo con il quale il parser richiede un altro token all'analizzatore lessicale;

- il front-end si occupa di trasformare il codice sorgente in codice tradotto nel linguaggio intermedio scelto; quindi fanno parte del front-end le fasi di compilazione descritte poco sopra.

Ci si aspetta che il parser riporti eventuali errori di sintassi se il programma sorgente *P* non rispetta le regole grammaticali del linguaggio in cui è scritto.

Il parsing può essere fatto in tre diversi metodi per l'analisi di grammatiche *context-free*:

1. Metodi di parsing universali come l'algoritmo di Cocke-Younger-Kasami o l'algoritmo di Earley: possono fare il parsing di un qualsiasi tipo di grammatica, ma proprio per la loro generalità possono essere inefficienti.

2. *Top-Down parsing*: l'albero di sintassi viene costruito dall'alto – ovvero dalla radice – fino al basso – ovvero le foglie, leggendo l'input da sinistra a destra.

3. *Bottom-Up parsing*: l'albero di sintassi viene costruito dal basso – ovvero le foglie – fino all'alto – ovvero la radice, leggendo l'input da sinistra a destra.

8.1.1 Top-Down parsing

Un parser *top-down* può essere visto come il tentativo di costruire un albero di derivazione per la stringa di input a partire dalla radice dell'albero espandendo i nodi dell'albero da sinistra a destra.

Ipotizzando di avere le seguenti regole di produzione della grammatica *G*, che generano un sottoinsieme delle stringhe che definiscono i tipi in Pascal:

type → *simple* | ***id*** | ***array*** [*simple*] ***of*** *type*

simple → ***integer*** | ***char*** | ***num dotdot num***

e la stringa da analizzare:

array* [*num dotdot num*] *of integer

Un parser *top-down* inizia a costruire l'albero dalla radice:

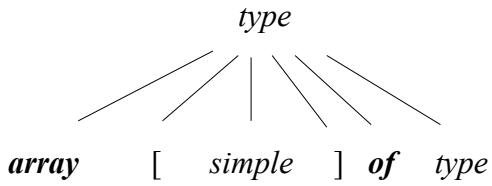
type

Ad ogni passo il parser sceglie come nodo da espandere quello etichettato con un non terminale che sia senza figli e che si trova più a sinistra nell'albero parziale; in questo caso l'unica scelta possibile è *type*. Una volta selezionato il nodo, il parser si deve basare sulla stringa di input per decidere come espanderlo: deve scegliere una regola di produzione che abbia come testa (alla sinistra della freccia) lo stesso nodo scelto; dopo di ciò deve aggiungere come figli di tale nodo il corpo della regola di produzione; se il corpo della regola è formato da tanti simboli, è necessario aggiungere all'albero tanti figli quanti sono i simboli.

In questo caso si è scelto la regola

type → ***array*** [*simple*] **of** *type*

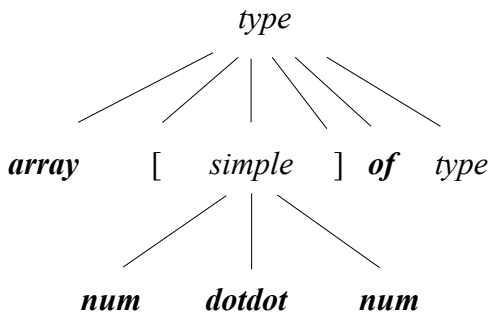
e l'albero parziale risultante è:



Come detto precedentemente, si sceglie il primo nodo sulla sinistra etichettato con un non terminale e senza figli: in questo caso *simple* e si espande con una delle regole di produzione la cui testa sia *simple*. In questo caso si è scelto la regola

simple → ***num dotdot num***

e l'albero risultante è

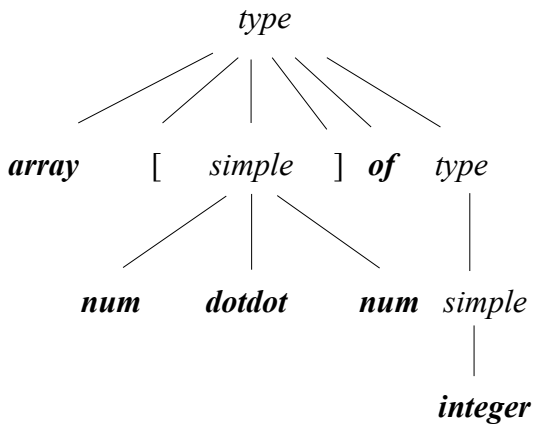


A questo punto si procede espandendo il nodo *type*, con le seguenti regole di produzione:

$$type \rightarrow simple$$

$$simple \rightarrow \textbf{integer}$$

L'albero risultante finale è il seguente



Il problema principale per un parser *top-down* è quello di scegliere correttamente con quale regola di produzione espandere il nodo selezionato. Se un albero di derivazione esiste per la stringa data come input e la grammatica non è ambigua, a ogni passo esiste solo una scelta corretta.

Un parser che torna sulle sue scelte se si rende conto che la strada che ha intrapreso non porta alla generazione della stringa data viene detto parser con *backtracking*. Mentre un parser che ad ogni passo sceglie la regola di produzione giusta si dice predittivo.

Come detto precedentemente, un parser *top-down* nella generazione dell'albero deve scegliere il nodo da espandere ma anche guardare la stringa di input per poter scegliere la regola di produzione giusta. Tuttavia il parser non guarda l'intera stringa, o ciò peserebbe troppo sull'efficienza del processo: deve guardare il numero minimo di token che gli servono per fare la scelta corretta. Il numero di simboli guardati vengono detti *lookahead* (letteralmente guardare avanti) e in genere è impostato a uno, anche se è possibile modificare questo valore.

Una grammatica per cui può essere scritto un parser predittivo è fatta in modo tale che, anche impostando il numero di *lookahead* a uno verrà scelta sempre la regola giusta: in sostanza perché ogni regola di produzione della grammatica ha un corpo che inizia con un simbolo diverso; ad esempio:

stmt → ***if*** *expr* ***then*** *stmt* ***else*** *stmt*

| ***while*** *expr* ***do*** *stmt*

| ***begin*** *stmt_list* ***end***

Evidentemente, se il primo simbolo dell'input è *if* allora il parser applicherà la prima produzione, se è *while* applicherà la seconda produzione e se è *begin* applicherà la terza.

8.1.2 Bottom-Up parsing

Una delle tecniche di analisi sintattica *bottom-up* prende il nome di *shift-reduce parsing* o analisi impila-riduci, che si occupa di costruire l'albero di derivazione della stringa di token che gli viene data in input a partire dalle foglie fino ad arrivare alla radice.

Questo processo può essere equiparato alla riduzione di una stringa *w* di simboli terminali in input al simbolo iniziale della grammatica. Ad ogni passo una sottostringa che corrisponde al corpo di una regola di produzione della grammatica, viene sostituita con la testa della regola in questione. Se ad ogni passo di riduzione la sottostringa viene scelta correttamente allora la sequenza di riduzioni fatte porta al simbolo iniziale della grammatica e la riduzione è andata a buon fine.

Ipotizzando di avere le seguenti regole di produzione:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

e consideriamo la stringa *abbcde.* Tale stringa può essere ridotta al simbolo iniziale *S*

tramite i seguenti passi (la sottostringa sottolineata è quella che viene ridotta):

a<u>b</u>bcde con la regola $A \rightarrow b$ diventa

a<u>Abc</u>de con la regola $A \rightarrow Abc$ diventa

aA<u>d</u>e con la regola $B \rightarrow d$ diventa

aABe con la regola $S \rightarrow aABe$ diventa

S

In sostanza bisogna cercare ad ogni passo una sottostringa uguale alla parte destra della regola di produzione. C'è tuttavia il problema che ne possano esistere più di una, ad esempio nel primo passo si poteva ridurre anche la seconda *e*. Dunque bisogna fare delle scelte.

8.2 Cosa è JavaCC e come funziona

Come detto precedentemente, il software utilizzato per l'implementazione del parser nel lavoro di sperimentazione svolto è JavaCC. Di seguito verrà illustrato più nel dettaglio.

JavaCC [26][27] è un generatore di parser e di analizzatore lessicale. Come già detto un analizzatore lessicale trasforma una sequenza di caratteri in delle sottosequenze chiamate token e le classifica.

Consideriamo un cortissimo programma scritto nel linguaggio di programmazione C:

```
int main() {

return 0 ;

}
```

L'analizzatore lessicale del compilatore C trasforma tale programma nella seguente sequenza di token:

```
“int”, “ ”, “main”, “(”, “)”,

“ ”, “{”, “\n”, “\t”, “return”

“ ”, “0”, “ ”, “;”, “\n”,

“}”, “\n”, “”
```

L'analizzatore inoltre identifica il tipo di ogni token: nell'esempio riportato la sequenza dei token è

```
KWINT, SPACE, ID, OPAR, CPAR,

SPACE, OBRACE, SPACE, SPACE, KWRETURN,

SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,

CBRACE, SPACE, EOF
```

Il token del tipo `EOF` rappresenta la fine del file originale. Tale sequenza è poi passata al parser. Nel caso del linguaggio di programmazione C il parser non ha bisogno di tutti i token: nell'esempio riportato il token classificato come `SPACE` non è considerato.

Dopo di ciò il parser analizza la sequenza di token per determinare la struttura del programma.

L'analizzatore lessicale e il parser, inoltre, sono responsabili della generazione di messaggi di errore, se il file di input non è conforme alle regole lessicali o sintattiche del linguaggio.

JavaCC non è un parser o un analizzatore lessicale ma un generatore di tali strumenti in Java. Ciò significa che genera un l'analizzatore e il parser in accordo alle specifiche che legge da un file.

Tali strumenti sono componenti complessi e per essere implementati in Java è necessario un lavoro lungo e complesso; con un generatore di parser si può risparmiare molto tempo e avere un lavoro di qualità.

Di seguito verrà mostrato un esempio di come funziona JavaCC per un semplice

programma di input come il seguente

```
99 + 42 + 0 + 15
```

Sono ammessi gli spazi ovunque eccetto tra i numeri. Inoltre gli unici caratteri ammessi nel programma di input possono essere le cifre da 0 a 9 e il segno dell'addizione.

Per implementare il parser per tale programma è necessario compilare un file con estensione .jj, tale file contiene le specifiche per il parser e l'analizzatore lessicale e sarà usato da JavaCC come input.

La prima parte di tale file sarà:

```
options {
   STATIC = false ;
}
PARSER BEGIN(Adder)
   class Adder {
      static void main( String[] args )
      throws ParseException, TokenMgrError {
         Adder parser = new Adder( System.in ) ;
         parser.Start() ; }
}
PARSER END(Adder)
```

Il file inizia con una parte in cui devono essere inserite tutte le opzioni: in questo caso l'unica opzione da "cambiare" è quella `STATIC` che di default è impostata come true. In questa sezione è possibile modificare il *lookahead* di cui si è parlato nella sezione 8.1.1, che di default è impostato a uno.

Di seguito un frammento di una classe Java chiamata Adder: non è presente tutta la classe, JavaCC aggiungerà le dichiarazioni alla classe come parte del processo generativo. Nel main sono dichiarate due eccezioni: `ParseException` e `TokenMgrError`.

Le specifiche dell'analizzatore lessicale di tale esempio sono le seguenti:

```
SKIP : { " " }
SKIP : { "\n" | "\r" | "\r\n" }
```

```
TOKEN : { < PLUS : "+" > }
TOKEN : { < NUMBER : (["0"-"9"])+ > }
```

La prima riga dice che il carattere spazio costituisce un token ma deve essere saltato e non passato al parser. La seconda riga fa lo stesso ma con gli a-capo. I caratteri con i quali sono espressi gli a-capo sono tre perché diversi sistemi operativi li rappresentano con diversi caratteri: i sistemi Unix e Linux li rappresentano con il carattere \n, i sistemi DOS e Windows con il carattere \r e i vecchi sistemi Macintoshe con il carattere \r\n. Tutte queste possibilità sono separate dalla barra verticale usata per le opzioni. La terza linea dice che il segno dell'addizione è un token e gli dà il nome simbolico di PLUS. Infine la quarta riga dice che i numeri sono dei token e dà loro il nome di NUMBER; la specifica dei NUMBER è espressa con un'espressione regolare $(['0' – '9'])^{+}$.

Ognuna di queste quattro righe è chiamata *regular expression production* (produzione di espressioni regolari).

Se venisse passata la stringa "5 – 3", l'analizzatore lessicale incontrando il carattere della sottrazione, che non è presente nei token, solleverebbe un'eccezione.

Se la stringa, invece, fosse " 5 5 + 3" oppure " 5 + + 3" l'eccezione sarebbe sollevata dal parser.

Le specifiche del parser sono scritte come delle produzioni BNF:

```
void Start() :
{}
{
   <NUMBER>
   (
      <PLUS>
      <NUMBER>
   )*
   <EOF>
}
```

La produzione specifica quale sia la sequenza corretta di token, ovvero la sequenza inizia con un token NUMBER e finisce con EOF (End Of File); tra questi due elementi possono esserci zero o più occorrenze di sequenze composte da un token PLUS e un token NUMBER.

A questo punto, per generare il parser e l'analizzatore lessicale bisogna invocare JavaCC sul file con estensione .jj e quest'azione genera sette classi Java:

1. *TokenMgr Error*: è una semplice classe di errori;

2. *ParseException*: è un'altra classe di errori;

3. *Token*: è una classe che rappresenta i token;

4. *SimpleCharStream*: è una classe che consegna i caratteri all'analizzatore lessicale;

5. *AdderConstants*: è un interfaccia che definisce il numero di classi usate sia nell'analizzatore lessicale che nel parser;

6. *AdderTokenManager*: è l'analizzatore lessicale;

7. *Adder*: è il parser.

Ritorniamo sul metodo main della classe adder. La prima dichiarazione del corpo crea un nuovo oggetto parser. Il costruttore usato è generato automaticamente e prende come parametro un *InputStream*. Tale costruttore restituisce le classi *SimpleCharStream* e *AdderTokenManager*: il parser riceve i token dall'analizzatore che legge i caratteri dall'oggetto *System.in* attraverso l'oggetto *SimpleCharacterStream*.

La seconda dichiarazione chiama un metodo chiamato *Start.* Per ogni produzione BNF nelle specifiche del file, JavaCC genera un metodo corrispondente nella classe del parser. Questo metodo è responsabile della ricerca nello stream di input di una corrispondenza per la descrizione dell'input. Nell'esempio fatto, il metodo *Start* ricerca una sequenza di token nell'input che corrisponda alla descrizione

```
<NUMBER> (<PLUS> <NUMBER>)* <EOF>
```

Nell'esempio appena fatto ci troviamo di fronte a un parser molto semplice in cui l'unica produzione BNF è quella del metodo *Start*. Se, però, dovesse essere descritto un linguaggio più complesso in cui sono presenti più di un non terminale del linguaggio, verrebbero prodotte tante produzioni quanti sono i non terminali.

Fatto ciò, si può sottoporre un file di input al parser e ci si può trovare in tre situazioni

differenti:

1. Viene trovato un errore lessicale: viene ritrovato dall'analizzatore un carattere inaspettato. Nell'esempio riportato, ciò avviene se si sottopone un file di tale tipo:

   123 – 456\n

   In questo caso il programma lancia un'eccezione come la seguente:

   Exception in thread “main” TokenMgrError: Lexical Error at line 1, column 5. Encountered: ”-” (45), after : ””

2. Viene trovato un *parsing error*: viene ritrovato dal parser quando una sequenza di token non è trovata nelle specifiche di *Start*. Nell'esempio riportato, ciò avviene se si sottopone un file di tale tipo:

   123 ++ 456

   In questo caso il programma lancia un'eccezione come la seguente:

   Exception in thread ”main” ParseException: Encountered ”+” at line 1, column 6. Was expecting: <NUMBER>

3. L'input contiene una sequenza di token che vengono ritrovati nelle specifiche di *Start*. In questo caso non viene lanciata nessuna eccezione e il programma semplicemente termina.

In sostanza, finché il parser non fa nulla quando l'input è legittimo, il suo uso è limitato a verificare la legittimità dei suoi input.

8.3 Risultati ottenuti dall'implementazione del parser e verifica sui modelli implementati

Come detto nel paragrafo precedente, per implementare il parser con JavaCC è necessario compilare un file con estensione .jj: tale file contiene le specifiche per il parser e l'analizzatore lessicale e sarà usato da JavaCC come input.

Nel caso del presente lavoro:

- è stato creato un file di nome Model.jj;
- è stato generato il parser da riga di comando invocando JavaCC su tale file[3]:

  ```
  javacc Model.jj
  ```

  con tale comando vengono generate sette classi Java;
- queste sette classi vengono compilate con il comando:

  ```
  javac *.java
  ```
- Infine è possibile far girare il programma sottoponendo un file di input al parser

  con tale comando:

  ```
  java Model <fileDiInput
  ```

Il file Model.jj è consultabile interamente nell'Appendice C. Di seguito verrà spiegato più nel dettaglio.

Il file inizia nel seguente modo:

```
options {
}
PARSER_BEGIN(Model)
public class Model {
  public static void main(String args[]) {
    Model parser;
    if (args.length == 0) {
      System.out.println("GameDSL Parser:  Reading from standard
input . . .");
      parser = new Model(System.in);
    } else if (args.length == 1) {
      System.out.println("GameDSL Parser:  Reading from file " +
args[0] + " . . .");
      try {
```

3 I comandi mostrati da qui in avanti sono quelli necessari per generare e utilizzare il parser sul sistema operativo Ubuntu.

```
        parser = new Model(new java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
        System.out.println("GameDSL Parser:  File " + args[0] + "
not found.");
        return;
      }
    } else {
      System.out.println("GameDSL Parser:  Usage is one of:");
      System.out.println("         java GameDSLParser <
inputfile");
      System.out.println("OR");
      System.out.println("         java GameDSLParser inputfile");
      return;
    }
    try {
      parser.Model();
      System.out.println("GameDSL Parser:  GameDSL program parsed
successfully.");
    } catch (ParseException e) {
      System.out.println("GameDSL Parser:  Encountered errors
during parse.");
      e.printStackTrace();
    }
  }
}
PARSER_END(Model)
```

La parte in cui sono specificate le opzioni del parser è vuota: ciò significa che tutte le impostazioni mantengono i valori di default di JavaCC. Dopo di ciò viene definito il parser, chiamato appunto Model, con un metodo main nel quale viene creato l'oggetto. Oltre a ciò si specifica in che modo è possibile sottoporre il file da parsare a Model: con il simbolo "<" prima del nome del file o senza di esso.

Inoltre vengono specificati i messaggi da stampare nel caso in cui il file venga parsato senza riscontrare nessun errore (`"GameDSL Parser:  GameDSL program parsed successfully."`) e nel caso in cui l'operazione di parsing non vada a buon fine (`"GameDSL Parser:  Encountered errors during parse."`).

Come già detto nel file con estensione .jj, è necessario identificare le specifiche per

l'analizzatore lessicale – ovvero delle liste di token, quelli che è necessario passare al parser e quelli che devono essere saltati – e le specifiche per il parser, scritte con notazione BNF, che devono in sostanza riprodurre le regole della grammatica del linguaggio.

Per quel che riguarda le specifiche per l'analizzatore lessicale, in Model.jj, troviamo una lista definita dalla parola “skip” e una lista definita dalla parola “special token”: nella prima lista sono presenti tutti i caratteri che non devono essere analizzati dal parser, ovvero spazi bianchi e caratteri di a-capo; nella seconda lista, invece, sono presenti le specifiche per l'inserimento dei commenti.

Troviamo quindi una lista di token che definiscono le parole chiave del linguaggio (come “init”, “class”, “termination”, “step” e via dicendo), una lista di token per i numeri e una lista di token per i caratteri (con la quale è possibile definire gli identificatori nel file da sottoporre al parser).

Le specifiche per il parser, come già detto, devono riprodurre le regole grammaticali di GameDSL. La prima regola che deve essere scritta è quella che rappresenterà la radice dell'albero di sintassi astratta: in questo caso Model. Tale regola deve contenere il token <EOF> per assicurarsi che venga parsato tutto il file.

Nella scrittura delle regole è necessario stare attenti ad un fondamentale aspetto che è quello dell'ambiguità che può verificarsi tra diverse regole di produzione. Vediamo più nel dettaglio di cosa si tratta.

Tra i non terminali di GameDSL troviamo la categoria sintattica delle espressioni. In questa categoria sintattica sono state raggruppate sia le espressioni aritmetiche che le espressioni booleane. Sarà poi compito di una successiva fase di compilazione analizzare e distinguere i diversi tipi di dati.

In questo raggruppamento, in particolare, sono state inserite le operazioni aritmetiche tra espressioni, le espressioni con operatori unari, le espressioni tra parentesi, le espressioni condizionali e le relazioni tra espressioni, che comprensibilmente hanno tutte precedenze diverse. Secondo tali precedenze il parser genera alberi di derivazione differenti.

La struttura di un albero di derivazione per una data stringa è molto importante nel contesto del parsing e dei compilatori in generale: se una frase di un linguaggio di programmazione viene strutturata in maniera corretta rispetto alla sua semantica, la

traduzione della stessa risulterà estremamente semplice. Si consideri ad esempio un linguaggio con la seguente grammatica *G*:

$E \rightarrow E + E \mid E * E \mid \mathbf{id}$

Questa grammatica è ambigua poiché per la stringa *3 + 5 * 4* possono essere generati due alberi di derivazione differenti e conseguentemente la stringa può produrre due risultati diversi, dal momento che ogni nodo interno di un albero di derivazione corrisponde a una sottoespressione il cui valore può essere calcolato in base ai valori associati ai nodi figli.

Di seguito viene mostrato l'albero di derivazione che equivale all'espressione in cui si da precedenza alla moltiplicazione: il risultato dell'espressione vale in questo caso 23.

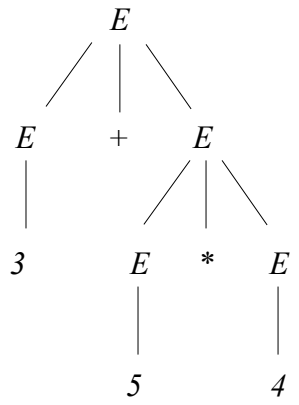


Illustrazione 8.1: Albero di derivazione per l'espressione

in cui si da precedenza alla moltiplicazione

Di seguito viene mostrato l'albero di derivazione che equivale all'espressione in cui si da precedenza alla somma: il risultato dell'espressione vale in questo caso 32.

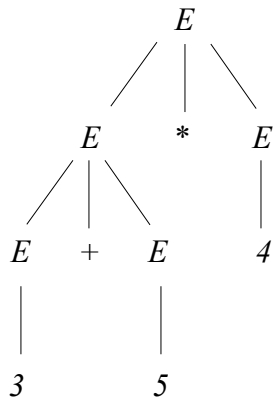


Illustrazione 8.2: Albero di derivazione per l'espressione

in cui si da precedenza alla somma

Una grammatica che non crea ambiguità deve, quindi, avere delle regole che risolvano l'ambiguità ponendo gli operatori su livelli diversi di precedenza [14].

Come detto precedentemente nella grammatica di GameDSL sono state raggruppate nella categoria sintattica delle espressioni tutte le produzioni, senza distinzione di precedenza. Al momento della creazione del parser, tuttavia, è necessario creare delle regole che siano ben annidate le une dentro le altre, in modo da non avere ambiguità e in modo che il parser possa essere generato senza problemi: se, infatti, tali regole generano ambiguità JavaCC non permette la creazione del parser.

Per questo motivo è stato necessario considerare la precedenza degli operatori, valutando quali sono gli operatori che "legano" di più (ovvero che devono essere valutati preliminarmente) per arrivare a quelli che "legano" di meno.

A tal proposito sono state definite le seguenti categorie sintattiche con rispettive regole di derivazione. Partendo dagli operatori che legano di meno troviamo:

- la regola per le espressioni condizionali:

```
void Expr():
{}
{
AndOrExp() ["?" Expr() ":" Expr()]
```

```
}
```

- la regole per le espressioni in AND e OR:

```
void AndOrExp():
{}
{
BExpr() ("&" BExpr() | "|" Bexpr())*
}
```

- la regola per gli operatori di confronto tra espressioni:

```
void Bexpr():
{}
{
AritExp()[(">" | "<" | "==" | "!=")AritExp()]
}
```

- la regola per la somma e la sottrazione tra espressioni:

```
void AritExp():
{}
{
Term() ("+" Term() | "-" Term())*
}
```

- la regola per la moltiplicazione e la divisione tra espressioni:

```
void Term():
{}
{
Unari() ( "*" Unari() | "/" Unari())*
}
```

- la regola per gli operatori unari:

```
void Unari():
{}
{
```

```
Primary() |"!" Primary() | "-" Primary()
}
```

- e, infine, la regola per gli elementi primari del linguaggio:

```
void Primary():
{}
{
<INT_LIT> | "(" [Expr()] ")" | "true" | "false"  | "random"
"("<INT_LIT> "," <INT_LIT>")"
| Pick()["."Path()] | "filter" "(" BasicType() <ID> ","
Expr() ")" "{"Expr()"}"["."<ID>] | "forall" "(" BasicType()
<ID> "," Expr() ")" "{"Expr()"}" | "exists" "(" BasicType()
<ID> "," Expr() ")" "{"Expr()"}" | Path()
}
```

Secondo tali regole una stringa come quella mostrata precedentemente (*3 + 5 * 4*) può produrre un solo albero di derivazione e conseguentemente un solo risultato: quello mostrato nell'Illustrazione 8.1, dal momento che preliminarmente viene valutata l'espressione *5 * 4* e solo in seguito viene valutata l'espressione *3 + 20*.

Oltre la questione della precedenza degli operatori, un altro fattore di ambiguità è la presenza di più regole di derivazione che iniziano nello stesso modo. In tale situazione il parser non sa effettivamente quale produzione applicare e genera un errore che arresta il processo di parsing.

Di seguito verrà mostrato un esempio e come si può risolvere la seguente situazione.

Nella categoria sintattica dei comandi si trovano, tra le altre, la regola di produzione per la dichiarazione di variabile:

```
CompoundType() <ID> "=" Expr() ";"
```

e la regola di produzione per l'assegnamento:

```
[Pick()"."]Path() ["=" Expr()]";"
```

dove `Path()` rappresenta un'ulteriore categoria sintattica in cui l'unica regola di

produzione è la seguente:

```
PathBase()["."Path()]
```

e `PathBase()`, a sua volta, rappresenta un'altra categoria sintattica in cui l'unica regola di produzione è la seguente:

```
<ID> [ "("[Expr() ("," Expr())*] ")" ] [ "[" Expr() "]" ]
```

È chiaro che, a seconda della regola di produzione scelta, verrà generato un albero di derivazione diverso.

Nel caso appena mostrato il parser non può essere generato per la presenza, appunto, di ambiguità tra le due regole di produzione: il terminale mostra, dunque, il seguente messaggio di errore:

```
Warning: Choice conflict involving two expansions at line 171,
         column 20 and line 171, column 58 respectively.
         A common prefix is: <ID>.
```

Per risolvere tale ambiguità è sufficiente impostare un *lookahead* diverso da quello di default, che come già detto è settato a uno. In questo caso è necessario impostarlo a due, che equivale a dire al parser che, quando deve definire un comando e si trova davanti un token di tipo `<ID>`, deve guardare due passi più avanti: in sostanza deve guardare cosa c'è dopo il token `<ID>` per capire quale regola di produzione scegliere.

Le due regole, dunque, vengono riscritte nel seguente modo:

```
void Com():
{}
{
LOOKAHEAD(2) CompoundType() <ID> "=" Expr() ";"
|  [Pick()"."]Path() ["=" Expr()]";"
}
```

Per la consultazione completa del file Model.jj si rimanda all'appendice C.

9 Conclusioni

In questo elaborato si è mostrato un quadro introduttivo ai concetti base necessari alla comprensione del lavoro: in particolare si è inteso fornire un background generale sui linguaggi formali, sulla sintassi del linguaggio e sui tre metodi con i quali è possibile definire tale linguaggio: la grammatica, gli automi e le espressioni regolari.

Si è cercato, inoltre, di inquadrare le tecniche di modellazione e model checking, tramite la spiegazione di alcuni modelli di sistemi transizionali: in particolare sono stati analizzati modelli di giochi, implementati in un precedente lavoro di sperimentazione. Tali analisi hanno permesso di capire in che modo le tecniche di modellazione e di model checking sono applicabili a diverse branche del sapere, quando non congeniali alla modellazione di sistemi ad hoc che permettono di verificare proprietà attese o desiderate.

Per la modellazione di tali giochi è stato utilizzato un software chiamato PRISM model checker. Le potenzialità di tale strumento hanno permesso di analizzare e quantificare alcuni aspetti, che nell'ambito del *game design*, è fondamentale considerare per poter produrre un buon gioco.

Nell'utilizzare tale strumento e nell'osservazione e analisi dei casi studio implementati ci si rende, tuttavia, subito conto di come il linguaggio di PRISM sia spesso inadeguato a descrivere in maniera concisa alcuni aspetti comuni dei giochi, come ad esempio l'ambiente di gioco, le strategie dei giocatori e la turnazione. In particolare l'aspetto più limitante del linguaggio di tale software è la presenza di pochi costrutti di programmazione e la quasi totale assenza di strutture dati, ad eccezione delle singole variabili.

È stato quindi deciso di creare un Domain Specific Language per la modellazione di giochi che rendesse la costruzione di modelli più semplice, efficace e concisa. Tuttavia, considerando il fatto che il software di model checking PRISM possiede numerose caratteristiche vantaggiose, come la verifica probabilistica o statistica di determinate proprietà espresse in logica formale, si presuppone una futura compilazione del linguaggio creato nel linguaggio di input di PRISM.

Il linguaggio creato si chiama GameDSL e ha sostanzialmente l'aspetto di un linguaggio di programmazione a oggetti, con liste e grafi come strutture dati primitive e una serie di costrutti di programmazione che permettono di descrivere in maniera concisa l'ambiente

di gioco, le strategie dei giocatori, la turnazione e specifici aspetti dei singoli casi studio che, con il linguaggio di PRISM, risultavano lunghi e complicati.

Per poter creare GameDSL è stato necessario definire una grammatica, con le diverse regole di produzione, e implementare un parser per l'analisi lessicale e sintattica dei modelli creati.

Per la creazione del parser è stato utilizzato il software JavaCC che consente di generare un parser con la scrittura di un semplice file di testo. In tale file, con estensione .jj, è necessario definire le specifiche per l'analisi lessicale e le specifiche per l'analisi sintattica, ovvero definire delle liste di token (parole) appartenenti al linguaggio e delle regole di produzione per le diverse categorie sintattiche della grammatica.

Una delle difficoltà riscontrate nella creazione del linguaggio riguarda la scelta di costrutti di programmazione e di strutture dati in grado di poter facilmente descrivere i giochi e le loro caratteristiche in maniera efficace e concisa ma che, in una futura compilazione nel linguaggio di PRISM, non producano eccessivi problemi.

Tenendo, ad esempio, in considerazione il fatto che, al momento della creazione delle variabili in PRISM non è più possibile eliminarle, si è deciso di creare delle strutture dati che avessero una struttura fissa e che potessero essere modificate solamente attraverso opportune operazioni di filtraggio. Lo stesso è avvenuto per i costrutti di programmazione, che sono stati creati tenendo conto delle possibili traduzioni nel linguaggio di PRISM.

Un'altra difficoltà tra quelle riscontrate è stata la definizione di regole grammaticali nelle quali fossero presenti produzioni che non creano problemi di ambiguità. Tale ambiguità si trova, ad esempio, nel momento in cui il parser non riesce a scegliere tra due o più produzioni diverse o nel momento in cui non viene gestita correttamente la precedenza degli operatori nella definizione delle espressioni.

Per quanto riguarda la prima problematica sono state definite regole grammaticali le cui produzioni non creano ambiguità nella scelta. Per risolvere, invece, il secondo problema sono state create regole grammaticali ben gerarchizzate che permettono la costruzione dell'albero di sintassi astratta senza generare ambiguità.

Una volta risolti tali problemi è stato possibile creare un linguaggio di alto livello, GameDSL, congeniale alla creazione di modelli di gioco in maniera concisa ed efficace, che si presuppone adatto a una futura traduzione nel linguaggio di input di PRISM.

Bibliografia

[1] E. M. Clarke Jr, O. Grumberg, D. Peled. Model checking. MIT press. 1999

[2] P. Milazzo, G. Pardini, D. Sestini, P. Bove. Case Studies of Application of Probabilistic and Statistical Model Checking in Game Design. In Games and Software Engineering (GAS), 2015 IEEE/ACM 4th International Workshop on IEEE 2015.

[3] D. Sestini. “Verifiche di proprietà di modelli di gioco con tecniche di model checking probabilistico” (pp. 13, 23-31, 59-60, 62, 79-80, 102-103). Disponibile a: https://etd.adm.unipi.it/t/etd-01192015-204228/;

[4] PRISM Model Checker. PRISM Downloads. PRISM-games. [on line]. Disponibile a: http://www.prismmodelchecker.org/games/; [consultato il 30/8/2015]

[5] The Model Checker Spin, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, (pp. 279-295).

[6] A. Cimatti, E. M. Clarke, et al. "Nusmv 2: An opensource tool for symbolic model checking." Computer Aided Verification. Springer Berlin Heidelberg, 2002.

[7] Oldenkamp, H. A. "Probabilistic model checking: A comparison of tools." 2007.

[8] Moreno-Ger, Pablo, et al. "Model-checking for adventure videogames." Information and Software Technology 51.3. 2009. (pp. 564-580).

[9] Shafiei, Navid, and Franck van Breugel. "Towards model checking of computer games with Java PathFinder." Games and Software Engineering (GAS), 2013 3rd International Workshop on. IEEE, 2013.

[10] Guana, Victor, and Eleni Stroulia. "Phydsl: A code-generation environment for 2d physics-based games." 2014 IEEE Games, Entertainment, and Media Conference (IEEE GEM). 2014.

[11] Smith, Adam M., Mark J. Nelson, and Michael Mateas. "Ludocore: A logical game engine for modeling videogames." Computational Intelligence and Games (CIG), 2010 IEEE Symposium on. IEEE, 2010.

[12] R. Barbuti, P. Mancarella, D. Pedreschi, F. Turini. Elementi di Sintassi dei Linguaggi di Programmazione . Appunti per gli studenti di Fondamenti di Programmazione. a.a. 2004/2005.

[13] G. Ausiello, F. d'Amore, G. Gambosi, L.Laura. Linguaggi, Modelli, Complessità (pp. 36-44). Franco Angelo Editore, 2014.

[14] L. Tesei. Dispense del corso di linguaggi di programmazione e compilatori. a.a. 2003/2004 (pp. 1-6, 21-22, 55-57, 66-70, 77-80, 92-93)

[15] Aiello, M., Albano A., Attardi G., Montanari U. Teoria della computabilità, logica, teoria dei linguaggi formali. ETS Pisa, 1976 (pp. 167-299).

[16] C. Baier, J. Katoen. Principles of Model Checking. MIT Press 2008.

[17] E. M. Clarke Jr, O. Grumberg, D. Peled. Model checking. MIT press. 1999

[18] E. M. Clarke, W. Klieber, M.˘s Nov´a˘cek, and P. Zuliani. Model checking and state explosion problem. In Tools for practical software verification. Volume 6659 of LNCS. (Pp. 1-30), Springer. 2012.

[19] PRISM Model Checker. PRISM Manual version 4.2. [on line]. Disponibile a: http://www.prismmodelchecker.org/manual/; [consultato il 10/9/2015]

[20] A. Deursen , P. Klint , J.Visser. Domain-Specific Languages: An Annotated Bibliography ARTICLE in ACM SIGPLAN NOTICES. JANUARY 2000

[21] M. Kwiatkowska, G. Norman, D. Parker. PRISM: probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Performance Evaluation Review, ACM, 2009, 36 (4), (pp.40-45).

[22] PRISM Model Checker. PRISM Case Studies [on line]. Disponibile a: http://www.prismmodelchecker.org/casestudies/index.php; [consultato il 9/9/2015]

[23] PRISM Model Checker. The dining philosopher problem [on line]. In PRISM Tutorial. Disponibile a: http://www.prismmodelchecker.org/tutorial/phil.php; [consultato il 9/9/2015]

[24] J. Huizinga. Homo ludens. Haarlem: Tijeenk Willing; 1938 (trad. it. Homo Ludens, Torino, Einaudi, 2002) (pag. 13)

[25] N. Fazio, A. Nicolosi e F. Barbanera. Introduzione alle Macchine Astratte. Sulla base delle note “Linguaggi ad Alto Livello, Macchine Astratte, Semantica” di P. Degano e G. Levi. Università di Pisa. [on line] Disponibile a: http://www.dmi.unict.it/~barba/Architetture.html/MATERIALE-IN-RETE/MarestrictedVersion/index.htm; [consultato il 11/9/2015]

[26] Java Compiler Compiler – The Java Parser Generator [on line]. Disponibile a: https://javacc.java.net/; [consultato il 15/9/2015]

[27] The JavaCC Tutorial [on line] Disponibile a: http://www.engr.mun.ca/~theo/JavaCC-Tutorial/; [consultato il 15/9/2015]

Appendice A: Modelli di gioco in GameDSL

A.1 Gioco dell'Oca

```
class Giocatore{
   int casella;
   void procedi(){
      casella = casella + random(1, 6);
      if(casella >63){
          casella = 63 - (casella – 63);
      }
   }
}

list Giocatori G ={(0), (0)};
Giocatore current_player=G[0];
int cTurni=0;
termination {exists (Giocatore g, G) {g.casella==63}}
step{
      current_player.procedi();
      current_player=G.next(current_player);
      cTurni=cTurni+1;
}
```

A.2 Risiko

```
distribution d(Territorio t, L) { 1/L.size }
class Giocatore {
   void posizionaArmate(int n) {
      pick(filter(Territorio t, A){t.proprietario==this}, d).armate
= n + pick(filter(Territorio t, A){t.proprietario==this},
d).armate;
   }
   int lanciaDado(){
      return random(1,6);
   }
   Territorio scegliChiAttaccare(){
      return pick(filter(Territorio t, A){t.proprietario!=this &
filter(Territorio t2, G.adj(t)){t2.proprietario==this &
t2.armate>1}.size > 0},d);
   }
   Territorio ScegliChiAttacca(Territorio dif){
      return pick(filter(Territorio t, G.adj(dif))
{t.proprietario==this && t.armate>1},d);
   }
   boolean attacca(){
      int r= random(0, 100);
      if(r>50 & filter(Territorio t, A){t.proprietario!=this &
filter(Territorio t2, G.adj(t)){t2.proprietario==this &
t2.armate>2}.size>0}.size>0){
         return true;
      }else{
         return false;
      }
   }
}
class Territorio {
   int armate;
   Giocatore proprietario;
}
```

```
list Giocatore P = {(),()};
list Territorio A = {(6,P[1]),(6,P[1]),(6,P[1]),(6,P[2]),(6,P[2]),
(6,P[2])};
edges E = {(0,1),(0,2),(1,3),(1,4),(2,3),(2,4),(3,5),(4,5)};
graph Territorio G = {A, E};
Giocatore current_player = P[0];
init{
     current_player.posizionaArmate(filter(Territorio t, A)
{t.proprietario==current_player}.size/2);
}
termination { forall (Territorio t, A) {t.proprietario ==
A[0].proprietario } }
step {
      if (current_player.attacca()) {
         Territorio dif = current_player.segliChiAttaccare();
         Territorio att = current_player.segliChiAttacca(dif);
         if (current_player.lanciaDado() >
dif.proprietario.lanciaDado()) {
            dif.armate = dif.armate - 1;
            if(dif.armate==0){
               dif.propietario=current_player;
               dif.armate=1;
               att.armate = att.armate - 1;
            }
         } else {
            att.armate = att.armate -1;
         }
      }else{
         current_player = P.next(current_player);
         current_player.posizionaArmate(filter(Territorio t, A)
{t.proprietario == current_player}.size/2);
      }
}
```

A.3 Dungeon Crawl

```
list Stanza A = {(random(4,7)), (random(4,7)), (random(4,7)),
(random(4,7)), (random(4,7)), (random(4,7)), (random(4,7)),
(random(4,7)), (random(4,7))};
list Giocatori G = {(7), (7), (7), (7)};
class Giocatore {
   int puntiFerita;
   int lanciaDado(){
      return random(1,6);
   }
}
class Mostro {
   int puntiFerita;
   int lanciaDado(){
      return random(1,6);
   }
}
class Stanza {
   list Mostro M = {(1), (1), (1), (1), (1), (1), (1)};
   int num_mostri;
   init(int n) {
      num_mostri = n;
   }
}
class StanzaFinale {
   int lanciaDadi() {
      return random(1,6) + random(1,6) + random(1,6) + random(1,6);
   }
}
boolean vinconoMostri = false;
boolean vinconoGiocatori = false;
Stanza current_room = A[0];
Giocatore current_player = G[0];
Mostro current_monster = current_room.M[0];
boolean stanzaFinale = false;
int lancioFinale = 0;
```

```
termination {vinconoGiocatori | vinconoMostri}
step {
   if (!stanzaFinale) {
      if (current_player.puntiFerita==0) {
         current_player = G.next(current_player);
      } else {
         if(current_player.lanciaDado() >
current_monster.lanciaDado()) {
            current_monster.puntiFerita =
current_monster.puntiFerita-1;
             if (current_monster.puntiFerita==0) {
               if(current_monster!
=current_room.M[current_room.M[num_mostri-1]]) {
                  M.next(current_monster);
               } else {
                  if(current_room!=A[8]) {
                     A.next(current_room);
                  } else {
                     stanzaFinale = true;
                     current_player=G[0];
                  }
               }
            }
         } else {
            current_player.puntiFerita =
current_player.puntiFerita-1;
            list Giocatore P = filter(Giocatore g, G)
{g.puntiFerita>0};
            if (P.size==0) {
               vinconoMostri=true;
            }
         }
      }
   } else {
      if (current_player.puntiFerita>0) {
         int lancio1 = current_player.lanciaDado();
        int lancio2 = current_player.lanciaDado();
         lancioFinale = lancioFinale + (lancio1 > lancio2 ? lancio1
: lancio2);
```

```
    }
    if (current_player==G[3]) {
       StanzaFinale sf = ();
       if (sf.lanciaDadi() > lancioFinale) {
          vinconoMostri=true;
       } else {
          vinconoGiocatori = true;
       }
    }
    current_player = G.next(current_player);
  }
}
```

A.4 Lotta a Squadre

```
distribution d(Giocatore g, L) { 1/L.size }
class Giocatore{
   int punti;
   int lanciaDado(){
      return random(1,6);
   }
   void attacca(Giocatore g2){
      g2.punti=g2.punti-lanciaDado();
   }
}
class Squadra{
   list Giocatore G={(10), (10), (10)};
   Giocatore scegliChiAttaccare(Squadra s){
      return pick(filter(Giocatore g, s.G){g.punti>0}, d);
   }
}
list Squadra S={(G), (G)};
Squadra current_team = S[0];
Giocatore current_player= current_team.G[0];
termination {exists(Squadra s, S) {forall(Giocatore g, s.G)
{g.punti==0}}}
step{
   if(current_player.punti>0){
      current_player.attacca ( current_team.scegliChiAttaccare
( S.next( current_team )));
   }
   if(current_player!=current_team.G[G.size-1]){
      current_player = current_team.G.next(current_player);
   }else{
      current_team = S.next(current_team);
      current_player = current_team.G[0];
   }
}
```

Appendice B: Modelli di gioco in PRISM

B.1 Gioco dell'Oca

```
dtmc
const int k;
const int player;
const double w;
// number of cells (STANDARD=63)
const int cell_count;
// who starts
const int initial_player = player;
// position of penalty cells (initialize to -1 to exclude them)
const int p1;
const int p2;
const int p3;
const int p4;
const int p5;
const int p6;
const int p7;
const int p8;
const int p9;
const int p10;
// maximum number of times a backwards move is applied
const int max_penalties = 10;
// number of backward steps applied when falling into a penalty
cell
const int backward_steps;
formula player1_wins = (x = cell_count);
formula player2_wins = (y = cell_count);
formula terminated = player1_wins | player2_wins;
module MTurns
next_step : [1..5] init (2 * initial_player) - 1;
[turn1a] !terminated & next_step = 1 -> (next_step' = 2);
```

```
[turn1b] !terminated & next_step = 2 -> (next_step' = 3);
[turn2a] !terminated & next_step = 3 -> (next_step' = 4);
[turn2b] !terminated & next_step = 4 -> (next_step' = 1);
[termination] terminated -> (next_step'=5);
endmodule
formula penalty_occurrence = cpx < max_penalties & (x = p1 | x = p2
| x = p3 | x = p4 | x = p5 | x = p6 | x = p7 | x = p8 | x = p9 | x
= p10);
module MPlayer1
x : [0..cell_count] init 0;
cpx : [0..max_penalties] init 0;
[turn1a] !terminated ->
    1/6 : (x' = (x + 1 <= cell_count) ? (x + 1) : (2 * cell_count -
x - 1)) +
    1/6 : (x' = (x + 2 <= cell_count) ? (x + 2) : (2 * cell_count -
x - 2)) +
    1/6 : (x' = (x + 3 <= cell_count) ? (x + 3) : (2 * cell_count -
x - 3)) +
    1/6 : (x' = (x + 4 <= cell_count) ? (x + 4) : (2 * cell_count -
x - 4)) +
    1/6 : (x' = (x + 5 <= cell_count) ? (x + 5) : (2 * cell_count -
x - 5)) +
    1/6 : (x' = (x + 6 <= cell_count) ? (x + 6) : (2 * cell_count -
x - 6)) ;
[turn1b] !terminated ->
    (x' = x - (penalty_occurrence ? backward_steps : 0)) &
    (cpx' = cpx + (penalty_occurrence ? 1 : 0));
endmodule
module MPlayer2 = MPlayer1 [x = y, cpx = cpy, turn1a = turn2a,
turn1b = turn2b, x_dice_counter = y_dice_counter] endmodule
rewards "turn_count"
  [turn1a] true : 1;
endrewards
rewards "player1_position"
  [termination] true : x;
endrewards
rewards "player2_position"
  [termination] true : y;
endrewards
```

```
rewards "second_player_position"
  [termination] true : min(x,y);
endrewards
system MTurns || MPlayer1 || MPlayer2 endsystem
```

B.2 Risiko

```
dtmc
const int tank_threshold_1;
const int tank_threshold_2;
const int turn_threshold_1;
const int turn_threshold_2;
// constants for identifying players
const int player1 = 0;
const int player2 = 1;
formula other_player = 1 - current_player;
// probability of attack for player1
const double p1;
// probability of attack forplayer2
const double p2;
// formulas for constant strategy
//formula prob1 =  p1;
//formula prob2 =  p2;
// formulas for tank dependent strategy
//formula prob1 = (total_tanks_p1 > tank_threshold_1 ? 0.9 : 0.1);
//formula prob2 = (total_tanks_p1 > tank_threshold_2 ? 0.9 : 0.1);
// formulas for turn dependent strategy
formula prob1 = (turn_counter > turn_threshold_1 ? 0.9 : 0.1);
formula prob2 = (turn_counter > turn_threshold_2 ? 0.9 : 0.1);
// formula to determine attack probability depending on player
attack parameter
formula attack_probability = (current_player = 0 ? prob1 : prob2);
// maximum number of tanks allowed in each territory
const int max_tanks = 18;
// maximum number of turns (to avoid infinite play)
const int max_turns = 1000;
formula total_tanks_p1 = c1 + c2 + c3;
formula total_tanks_p2 = c4 + c5 + c6;
formula total_tanks = total_tanks_p1 + total_tanks_p2;
// termination
formula player1_wins = s1 = 0 & s2 = 0 & s3 = 0 & s4 = 0 & s5 = 0 &
```

```
s6 = 0;
formula player2_wins = s1 = 1 & s2 = 1 & s3 = 1 & s4 = 1 & s5 = 1 &
s6 = 1;
formula terminating = player1_wins | player2_wins |
turn_counter=max_turns;
formula terminated = phase=6;
// boolean values (actually integers) to indicate if the current
player has at least 2 tanks in a territory
formula cs1 = (s1 = current_player & c1 >= 2) ? 1 : 0;
formula cs2 = (s2 = current_player & c2 >= 2) ? 1 : 0;
formula cs3 = (s3 = current_player & c3 >= 2) ? 1 : 0;
formula cs4 = (s4 = current_player & c4 >= 2) ? 1 : 0;
formula cs5 = (s5 = current_player & c5 >= 2) ? 1 : 0;
formula cs6 = (s6 = current_player & c6 >= 2) ? 1 : 0;
// number of territories owned by each player
formula p1_territories_count = (1 - s1) + (1 - s2) + (1 - s3) + (1
- s4) + (1 - s5) + (1 - s6);
formula p2_territories_count = s1 + s2 + s3 + s4 + s5 + s6;
// number of territories owned by the current player
formula territories_count = current_player = player1 ?
p1_territories_count : p2_territories_count;
// number of reinforcements (as tanks) given to the current player
at the beginning of its turn
formula reinforcements = floor(territories_count / 2);
// boolean values to indicate if the current player owns a
territory
formula current_player_owns_s1 = (current_player = player1 ? (1 -
s1) : s1);
formula current_player_owns_s2 = (current_player = player1 ? (1 -
s2) : s2);
formula current_player_owns_s3 = (current_player = player1 ? (1 -
s3) : s3);
formula current_player_owns_s4 = (current_player = player1 ? (1 -
s4) : s4);
formula current_player_owns_s5 = (current_player = player1 ? (1 -
s5) : s5);
formula current_player_owns_s6 = (current_player = player1 ? (1 -
s6) : s6);
// probabilities of adding reinforcements to each territory owned
```

```
by the current player
// (value is 0 for territories *not* owned by the current player)
formula reinforcements_s1 = current_player_owns_s1 /
territories_count;
formula reinforcements_s2 = current_player_owns_s2 /
territories_count;
formula reinforcements_s3 = current_player_owns_s3 /
territories_count;
formula reinforcements_s4 = current_player_owns_s4 /
territories_count;
formula reinforcements_s5 = current_player_owns_s5 /
territories_count;
formula reinforcements_s6 = current_player_owns_s6 /
territories_count;
// boolean values to indicate if the other player owns a territory
formula other_player_owns_s1 = 1 - current_player_owns_s1;
formula other_player_owns_s2 = 1 - current_player_owns_s2;
formula other_player_owns_s3 = 1 - current_player_owns_s3;
formula other_player_owns_s4 = 1 - current_player_owns_s4;
formula other_player_owns_s5 = 1 - current_player_owns_s5;
formula other_player_owns_s6 = 1 - current_player_owns_s6;
// boolean values to indicate if the current player can attack a
territory
formula can_attack_s1 = other_player_owns_s1 * max(cs2, cs3);
formula can_attack_s2 = other_player_owns_s2 * max(cs1, cs4, cs5);
formula can_attack_s3 = other_player_owns_s3 * max(cs1, cs4, cs5);
formula can_attack_s4 = other_player_owns_s4 * max(cs2, cs3, cs6);
formula can_attack_s5 = other_player_owns_s5 * max(cs2, cs3, cs6);
formula can_attack_s6 = other_player_owns_s6 * max(cs4, cs5);
// number of territories from which an attack can be launched by
the current player
formula can_attack_count = can_attack_s1 + can_attack_s2 +
can_attack_s3 + can_attack_s4 + can_attack_s5 + can_attack_s6;
// boolean values to indicate if a territory is the *only* one with
no tanks
formula only_s1_empty = c1 = 0 & min(c2, c3, c4, c5, c6) > 0;
formula only_s2_empty = c2 = 0 & min(c1, c3, c4, c5, c6) > 0;
formula only_s3_empty = c3 = 0 & min(c1, c2, c4, c5, c6) > 0;
formula only_s4_empty = c4 = 0 & min(c1, c2, c3, c5, c6) > 0;
```

```
formula only_s5_empty = c5 = 0 & min(c1, c2, c3, c4, c6) > 0;
formula only_s6_empty = c6 = 0 & min(c1, c2, c3, c4, c5) > 0;
// if there is some territory with no tanks
formula some_empty_territory = (c1 = 0 | c2 = 0 | c3 = 0 | c4 = 0 |
c5 = 0 | c6 = 0);
module MPlayer
turn_counter: int init 0;
// who plays in the current turn; it alternates between 0 (player1)
and 1 (player2)
current_player : [0..1] init 0;
// who owns each territory (0: first player, 1: second player)
s1 : [0..1] init 0;
s2 : [0..1] init 0;
s3 : [0..1] init 0;
s4 : [0..1] init 1;
s5 : [0..1] init 1;
s6 : [0..1] init 1;
// number of tanks in each territory
c1 : [0..max_tanks] init 6;
c2 : [0..max_tanks] init 6;
c3 : [0..max_tanks] init 6;
c4 : [0..max_tanks] init 6;
c5 : [0..max_tanks] init 6;
c6 : [0..max_tanks] init 6;
// current phase during a player's turn
phase : [0..6] init 0;
// the territory selected as the *source* of an attack (owned by
the current player)
att : [0..6] init 0;
// the territory selected as the *target* of an attack (owned by
the other player)
dif : [0..6] init 0;
// extra final transition necessary to properly compute rewards
[] terminating -> (phase'=6);
// phase 0: the current player receives its reinforcements
[] phase = 0 & !terminating & territories_count > 0 ->
    reinforcements_s1 : (c1' = min(c1 + reinforcements, max_tanks))
& (phase' = 1) & (turn_counter'=turn_counter+1) +
    reinforcements_s2 : (c2' = min(c2 + reinforcements, max_tanks))
```

```
& (phase' = 1) & (turn_counter'=turn_counter+1)+
    reinforcements_s3 : (c3' = min(c3 + reinforcements, max_tanks))
& (phase' = 1) & (turn_counter'=turn_counter+1)+
    reinforcements_s4 : (c4' = min(c4 + reinforcements, max_tanks))
& (phase' = 1) & (turn_counter'=turn_counter+1)+
    reinforcements_s5 : (c5' = min(c5 + reinforcements, max_tanks))
& (phase' = 1) & (turn_counter'=turn_counter+1)+
    reinforcements_s6 : (c6' = min(c6 + reinforcements, max_tanks))
& (phase' = 1) & (turn_counter'=turn_counter+1);
// phase 1: randomly decide to attack (phase 2), or let turn change
(back to phase 0)
[attack_count] phase = 1 & !terminating ->
          attack_probability : (phase' = 2) +  // attack
      1 - attack_probability : (phase' = 0) & (current_player' =
other_player); // change turn
// phase 2, when some territory can be attacked: determine which
territory to attack (dif)
[] phase = 2 & can_attack_count > 0 ->
      can_attack_s1/can_attack_count : (dif' = 1) & (phase' = 3) +
      can_attack_s2/can_attack_count : (dif' = 2) & (phase' = 3) +
      can_attack_s3/can_attack_count : (dif' = 3) & (phase' = 3) +
      can_attack_s4/can_attack_count : (dif' = 4) & (phase' = 3) +
      can_attack_s5/can_attack_count : (dif' = 5) & (phase' = 3) +
      can_attack_s6/can_attack_count : (dif' = 6) & (phase' = 3);
// phase 2, when no territory can be attacked: change turn (back to
phase 0)
[] phase = 2 & can_attack_count = 0 ->
      (phase' = 0) & (current_player' = other_player);  // change
turn
// phase 3: determine from which territory the attack is launched
(att)
// case dif = 1
[] phase = 3  & dif = 1 & cs2 + cs3 > 0 ->
      cs2/(cs2 + cs3) : (att' = 2) & (phase' = 4) +
      cs3/(cs2 + cs3) : (att' = 3) & (phase' = 4);
// case dif = 2/3
[] phase = 3  & (dif = 2 | dif = 3) & cs1 + cs4 + cs5 > 0 ->
      cs1/(cs1 + cs4 + cs5) : (att' = 1) & (phase' = 4) +
      cs4/(cs1 + cs4 + cs5) : (att' = 4) & (phase' = 4) +
```

```
      cs5/(cs1 + cs4 + cs5) : (att' = 5) & (phase' = 4);
// case dif = 4/5
[] phase = 3  & (dif = 4 | dif = 5) & cs2 + cs3 + cs6 > 0 ->
      cs2/(cs2 + cs3 + cs6) : (att' = 2) & (phase' = 4) +
      cs3/(cs2 + cs3 + cs6) : (att' = 3) & (phase' = 4) +
      cs6/(cs2 + cs3 + cs6) : (att' = 6) & (phase' = 4);
// case dif = 6
[] phase = 3  & dif = 6 & cs4 + cs5 > 0 ->
      cs4/(cs4 + cs5) : (att' = 4) & (phase' = 4) +
      cs5/(cs4 + cs5) : (att' = 5) & (phase' = 4);
// phase 4: for each pair att/dif, model the possible outcomes of
an attack
// (attacker wins with probability 0.417, defender with probability
0.583)
[] phase = 4 & dif = 1 & att = 2 & !some_empty_territory -> 0.417 :
(c1' = c1 - 1) & (phase' = 5) +  0.583 : (c2' = c2 - 1) & (phase' =
5);
[] phase = 4 & dif = 1 & att = 3 & !some_empty_territory -> 0.417 :
(c1' = c1 - 1) & (phase' = 5) +  0.583 : (c3' = c3 - 1) & (phase' =
5);
[] phase = 4 & dif = 2 & att = 1 & !some_empty_territory -> 0.417 :
(c2' = c2 - 1) & (phase' = 5) +  0.583 : (c1' = c1 - 1) & (phase' =
5);
[] phase = 4 & dif = 2 & att = 4 & !some_empty_territory -> 0.417 :
(c2' = c2 - 1) & (phase' = 5) +  0.583 : (c4' = c4 - 1) & (phase' =
5);
[] phase = 4 & dif = 2 & att = 5 & !some_empty_territory -> 0.417 :
(c2' = c2 - 1) & (phase' = 5) +  0.583 : (c5' = c5 - 1) & (phase' =
5);
[] phase = 4 & dif = 3 & att = 1 & !some_empty_territory -> 0.417 :
(c3' = c3 - 1) & (phase' = 5) +  0.583 : (c1' = c1 - 1) & (phase' =
5);
[] phase = 4 & dif = 3 & att = 4 & !some_empty_territory -> 0.417 :
(c3' = c3 - 1) & (phase' = 5) +  0.583 : (c4' = c4 - 1) & (phase' =
5);
[] phase = 4 & dif = 3 & att = 5 & !some_empty_territory -> 0.417 :
(c3' = c3 - 1) & (phase' = 5) +  0.583 : (c5' = c5 - 1) & (phase' =
5);
[] phase = 4 & dif = 4 & att = 2 & !some_empty_territory -> 0.417 :
```

```
(c4' = c4 - 1) & (phase' = 5) +  0.583 : (c2' = c2 - 1) & (phase' =
5);
[] phase = 4 & dif = 4 & att = 3 & !some_empty_territory -> 0.417 :
(c4' = c4 - 1) & (phase' = 5) +  0.583 : (c3' = c3 - 1) & (phase' =
5);
[] phase = 4 & dif = 4 & att = 6 & !some_empty_territory -> 0.417 :
(c4' = c4 - 1) & (phase' = 5) +  0.583 : (c6' = c6 - 1) & (phase' =
5);
[] phase = 4 & dif = 5 & att = 2 & !some_empty_territory -> 0.417 :
(c5' = c5 - 1) & (phase' = 5) +  0.583 : (c2' = c2 - 1) & (phase' =
5);
[] phase = 4 & dif = 5 & att = 3 & !some_empty_territory -> 0.417 :
(c5' = c5 - 1) & (phase' = 5) +  0.583 : (c3' = c3 - 1) & (phase' =
5);
[] phase = 4 & dif = 5 & att = 6 & !some_empty_territory -> 0.417 :
(c5' = c5 - 1) & (phase' = 5) +  0.583 : (c6' = c6 - 1) & (phase' =
5);
[] phase = 4 & dif = 6 & att = 4 & !some_empty_territory -> 0.417 :
(c6' = c6 - 1) & (phase' = 5) +  0.583 : (c4' = c4 - 1) & (phase' =
5);
[] phase = 4 & dif = 6 & att = 5 & !some_empty_territory -> 0.417 :
(c6' = c6 - 1) & (phase' = 5) +  0.583 : (c5' = c5 - 1) & (phase' =
5);
 // phase 5, if no territory has become empty: the current player
may continue playing (no turn change; back to phase 1)
[] phase = 5 & !some_empty_territory -> (dif' = 0) & (att' = 0) &
(phase' = 1);
// phase 5, if a territory has become empty: move one tank from the
attack territory to the conquered one, which becomes owned by the
current player
// the current player may continue playing (no turn change; back to
phase 1)
[] phase = 5 & att = 2 & only_s1_empty -> (c1' = c1 + 1) & (c2' =
c2 - 1) & (s1' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 3 & only_s1_empty -> (c1' = c1 + 1) & (c3' =
c3 - 1) & (s1' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 1 & only_s2_empty -> (c2' = c2 + 1) & (c1' =
```

```
c1 - 1) & (s2' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 4 & only_s2_empty -> (c2' = c2 + 1) & (c4' =
c4 - 1) & (s2' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 5 & only_s2_empty -> (c2' = c2 + 1) & (c5' =
c5 - 1) & (s2' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 1 & only_s3_empty -> (c3' = c3 + 1) & (c1' =
c1 - 1) & (s3' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 4 & only_s3_empty -> (c3' = c3 + 1) & (c4' =
c4 - 1) & (s3' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 5 & only_s3_empty -> (c3' = c3 + 1) & (c5' =
c5 - 1) & (s3' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 2 & only_s4_empty -> (c4' = c4 + 1) & (c2' =
c2 - 1) & (s4' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 3 & only_s4_empty -> (c4' = c4 + 1) & (c3' =
c3 - 1) & (s4' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 6 & only_s4_empty -> (c4' = c4 + 1) & (c6' =
c6 - 1) & (s4' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 2 & only_s5_empty -> (c5' = c5 + 1) & (c2' =
c2 - 1) & (s5' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 3 & only_s5_empty -> (c5' = c5 + 1) & (c3' =
c3 - 1) & (s5' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 6 & only_s5_empty -> (c5' = c5 + 1) & (c6' =
c6 - 1) & (s5' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 4 & only_s6_empty -> (c6' = c6 + 1) & (c4' =
c4 - 1) & (s6' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
[] phase = 5 & att = 5 & only_s6_empty -> (c6' = c6 + 1) & (c5' =
```

```
c5 - 1) & (s6' = current_player) & (dif' = 0) & (att' = 0) &
(phase' = 1);
endmodule
rewards "turns"
      terminating & turn_counter<max_turns : turn_counter;
endrewards
rewards "attacks"
      [attack_count] true : 1;
endrewards
```

B.3 Dungeon Crawl

```
dtmc
// constants for identifying players
const int player1 = 1;
const int player2 = 2;
const int player3 = 3;
const int player4 = 4;
formula next_player = 1 + mod(current_player, 4);
const int alive;
const int current_room;
const int max_health;
const int initial_extra_health;
// computed with Dice.prism
const double player_wins_prob = 0.7453703703703705;
formula game_won = step = 8 & players_sum >= monster_sum;
formula game_lost = (step = 8 & players_sum < monster_sum) | (g1 =
0 & g2 = 0 & g3 = 0 & g4 = 0);
formula terminated = game_won | game_lost;
formula cp_health =
      (current_player = player1 ? g1 : 0) +
      (current_player = player2 ? g2 : 0) +
      (current_player = player3 ? g3 : 0) +
      (current_player = player4 ? g4 : 0);
formula cp_decrease_health = cp_health > 1 | (cp_health = 1 &
extra_health = 0);
formula cp_gets_extra_health = cp_health = 1 & extra_health > 0;
formula cp_lost_update = (cp_decrease_health ? cp_health - 1 : 0) +
(cp_gets_extra_health ? max_health : 0);
formula cp_lost_update_g1 = current_player = player1 ?
cp_lost_update : g1;
formula cp_lost_update_g2 = current_player = player2 ?
cp_lost_update : g2;
formula cp_lost_update_g3 = current_player = player3 ?
cp_lost_update : g3;
formula cp_lost_update_g4 = current_player = player4 ?
```

```
cp_lost_update : g4;
formula alive_players_count = (g1 > 0 ? 1 : 0) + (g2 > 0 ? 1 : 0) +
(g3 > 0 ? 1 : 0) + (g4 > 0 ? 1 : 0);
formula dead_players_count = 4 - alive_players_count;
const double recovery_probability;
formula cp_rescues_p1_prob = (current_player != player1 & cp_health
>= 3 ? 1 : 0) * recovery_probability * (g1 = 0 ?
1/dead_players_count : 0);
formula cp_rescues_p2_prob = (current_player != player2 & cp_health
>= 3 ? 1 : 0) * recovery_probability * (g2 = 0 ?
1/dead_players_count : 0);
formula cp_rescues_p3_prob = (current_player != player3 & cp_health
>= 3 ? 1 : 0) * recovery_probability * (g3 = 0 ?
1/dead_players_count : 0);
formula cp_rescues_p4_prob = (current_player != player4 & cp_health
>= 3 ? 1 : 0) * recovery_probability * (g4 = 0 ?
1/dead_players_count : 0);
formula cp_attack_prob = 1 - (cp_rescues_p1_prob +
cp_rescues_p2_prob + cp_rescues_p3_prob + cp_rescues_p4_prob);
formula rescue_health = (rescue = player1 ? g1 : 0) + (rescue =
player2 ? g2 : 0) + (rescue = player3 ? g3 : 0) + (rescue = player4
? g4 : 0);
formula cp_rescues_update_g1 = g1 + (current_player = player1 ?
-2 : 0) + (rescue = player1 ? 1 : 0);
formula cp_rescues_update_g2 = g2 + (current_player = player2 ?
-2 : 0) + (rescue = player2 ? 1 : 0);
formula cp_rescues_update_g3 = g3 + (current_player = player3 ?
-2 : 0) + (rescue = player3 ? 1 : 0);
formula cp_rescues_update_g4 = g4 + (current_player = player4 ?
-2 : 0) + (rescue = player4 ? 1 : 0);
module MGame
// who plays in the current turn
current_player : [1..4] init 1;
rescue: [-1..4] init -1;
room: [1..10] init 1;
g1: [0..max_health] init max_health;
g2: [0..max_health] init max_health;
g3: [0..max_health] init max_health;
g4: [0..max_health] init max_health;
```

```
extra_health: [-1..initial_extra_health] init initial_extra_health;
monsters: [-1..7] init -1;
turn: [1..4] init 1;
// 0: attack a monster; 1: rescue another player
choice: [-1..1] init -1;
[room_enter] room < 10 & monsters = -1 ->
      1/4: (monsters' = 4) +
      1/4: (monsters' = 5) +
      1/4: (monsters' = 6) +
      1/4: (monsters' = 7);
[] room < 10 & monsters > 0 & cp_health = 0 & !terminated ->
(current_player' = next_player);
[turn_c] room < 10 & monsters > 0 & cp_health > 0 & choice = -1 ->
      cp_attack_prob : (choice' = 0) +
      cp_rescues_p1_prob : (choice' = 1) & (rescue' = player1) +
      cp_rescues_p2_prob : (choice' = 1) & (rescue' = player2) +
      cp_rescues_p3_prob : (choice' = 1) & (rescue' = player3) +
      cp_rescues_p4_prob : (choice' = 1) & (rescue' = player4);
// attack
[] room < 10 & monsters > 0 & cp_health > 0 & choice = 0 ->
          player_wins_prob :
            (monsters' = monsters - 1) &
            (choice' = -1) &
            (current_player' = next_player) +
      1 - player_wins_prob :
            (g1' = cp_lost_update_g1) &
            (g2' = cp_lost_update_g2) &
            (g3' = cp_lost_update_g3) &
            (g4' = cp_lost_update_g4) &
            (extra_health' = extra_health - (cp_gets_extra_health ?
1 : 0)) &
            (choice' = -1) &
            (current_player' = next_player);
// current player rescues someone (variable 'rescue')
[] room < 10 & monsters > 0 & cp_health >= 3 & rescue_health = 0 &
choice = 1 ->
            (g1' = cp_rescues_update_g1) &
            (g2' = cp_rescues_update_g2) &
            (g3' = cp_rescues_update_g3) &
```

```
            (g4' = cp_rescues_update_g4) &
            (rescue' = -1) &
            (choice' = -1) &
            (current_player' = next_player);
[room_exit] room < 10 & monsters = 0 -> (room' = room + 1) &
(monsters' = -1) & (current_player' = room < 9 ? current_player :
1);
endmodule
module MFinalMonsterSteps
step: [-1..8] init -1;
[] room = 10 & step = -1 -> (step' = 0);
[phase1] 0 <= step & step < 4 -> (step' = step + 1);
[phase2] 4 <= step & step < 4 + alive_players_count -> (step' =
step + 1);
[] step = 4 + alive_players_count -> (step' = 8);
endmodule
module MFinalMonster
monster_sum: [0..6*4] init 0;
players_sum: [0..6*4] init 0;
[phase1] monster_sum <= 6*3 ->
      1/6: (monster_sum' = monster_sum + 1) +
      1/6: (monster_sum' = monster_sum + 2) +
      1/6: (monster_sum' = monster_sum + 3) +
      1/6: (monster_sum' = monster_sum + 4) +
      1/6: (monster_sum' = monster_sum + 5) +
      1/6: (monster_sum' = monster_sum + 6);
[phase2] players_sum <= 6*3 ->
       1/36: (players_sum' = players_sum + 1) +
       3/36: (players_sum' = players_sum + 2) +
       5/36: (players_sum' = players_sum + 3) +
       7/36: (players_sum' = players_sum + 4) +
       9/36: (players_sum' = players_sum + 5) +
      11/36: (players_sum' = players_sum + 6);
endmodule
module MChecker
allEnter_flag : bool init false;
[room_enter] true -> (allEnter_flag'= alive_players_count=4 );
[room_exit] true -> (allEnter_flag'= false );
endmodule
```

```
rewards "first_died"
      [room_enter] alive_players_count=4 : 1;
endrewards
rewards "turn_counter"
      [turn_c] true : 1;
endrewards
rewards "turn_counter_notAll"
      [turn_c] alive_players_count<4 : 1;
endrewards
```

B.4 Lotta a Squadre

```
dtmc
formula squadra1_wins = a=0 & b=0 & c=0;
formula squadra2_wins = x=0 & y=0 & z=0;
formula terminato = squadra1_wins | squadra2_wins;
formula puntiPersiA = 10 - a;
formula puntiPersiB = 10 - b;
formula puntiPersiC = 10 - c;
formula attaccoA = (puntiPersiA = 10) ? 0 : puntiPersiA+5;
formula attaccoB = (puntiPersiB = 10) ? 0 : puntiPersiB+5;
formula attaccoC = (puntiPersiC = 10) ? 0 : puntiPersiC+5;
formula xVivo = (x>0)?1:0;
formula yVivo = (y>0)?1:0;
formula zVivo = (z>0)?1:0;
formula aVivo = (a>0)?1:0;
formula bVivo = (b>0)?1:0;
formula cVivo = (c>0)?1:0;
formula numVivi = xVivo + yVivo + zVivo;
formula numVivi2 = aVivo + bVivo + cVivo;
formula perdeUnoPrimo = (numVivi=2)?2:((numVivi2=2)?1:0);
formula PerdenteX= ((x<y & x<z & x>0) | (y=0 & x<z & x>0) | (z=0 &
x<y & x>0) | (y=0 & z=0 & x>0))? numVivi:0;
formula PerdenteY= ((y<x & y<z & y>0) | (x=0 & y<z & y>0) | (z=0 &
y<x & y>0) | (x=0 & z=0 & y>0))? numVivi:0;
formula PerdenteZ= ((z<x & z<y & z>0) | (x=0 & z<y & z>0) | (y=0 &
z<x & z>0) | (x=0 & y=0 & z>0))? numVivi:0;
formula tuttiUguali1 = ((a=b & a=c & a>0) | (a=b & c=0) | (a=c &
b=0) | (b=c & a=0));
formula tuttiUguali2 = ((x=y & x=z & x>0) | (x=y & z=0) | (x=z &
y=0) | (y=z & x=0));
formula anomalo2= PerdenteX+PerdenteY+PerdenteZ=0 & !tuttiUguali2;
const int k;
const int giocatoreIniziale;
module MTurni
next_step : [0..3] init giocatoreIniziale;
```

```
[turno1] !terminato & next_step = 0 ->( next_step'=1);
[fineTurno1] !terminato & next_step=1 -> (next_step'=2);
[turno2] !terminato & next_step = 2 -> (next_step' = 3);
[fineTurno2] !terminato & next_step=3-> (next_step'=0);
endmodule
module MSquadra1 //modulo con strategia 1 (si attacca con
probabilità variabile a seconda dei punti persi)
turnoSq1: [0..2] init 0;
dado1 : [0..10] init 0;
//variabili dei giocatori della squadra2
a: [0..10] init 10;
b: [0..10] init 10;
c: [0..10] init 10;
//lancio del giocatore A
[turno1] !terminato & turnoSq1=0 & x>0->
                                        ((k>=1)?1:0)/k : (dado1'=1) +
                                        ((k>=2)?1:0)/k : (dado1'=2) +
                                        ((k>=3)?1:0)/k : (dado1'=3) +
                                        ((k>=4)?1:0)/k : (dado1'=4) +
                                        ((k>=5)?1:0)/k : (dado1'=5) +
                                        ((k>=6)?1:0)/k : (dado1'=6) +
                                        ((k>=7)?1:0)/k : (dado1'=7) +
                                        ((k>=8)?1:0)/k : (dado1'=8) +
                                        ((k>=9)?1:0)/k : (dado1'=9) +
                                       ((k>=10)?1:0)/k : (dado1'=10);
//se il giocatore A ha perso passa il turno al giocatore B
[turno1] !terminato & turnoSq1=0 & dado1=0 & x<=0-> (turnoSq1'=1);
//con probabilità 1/3 attacca uno dei tre della squadra1 (se hanno
gli stessi punti)
[]!terminato & turnoSq1=0 & dado1!=0 & tuttiUguali1->
                      1/3 : (a'= max(a-dado1, 0)) & (turnoSq1'=1)
& (dado1'=0)+
                      1/3 : (b'= max(b-dado1, 0)) & (turnoSq1'=1)
& (dado1'=0)+
                      1/3 : (c'= max(c-dado1, 0)) & (turnoSq1'=1)
& (dado1'=0);
//se non hanno gli stessi punti usa la strategia 2
[]!terminato & turnoSq1=0 & dado1!=0 & !tuttiUguali1->
                      attaccoA/(attaccoA+attaccoB+attaccoC)   :
```

```
(a'= max(a-dado1, 0)) & (turnoSq1'=1) & (dado1'=0)+
                        attaccoB/(attaccoA+attaccoB+attaccoC)  :
(b'= max(b-dado1, 0)) & (turnoSq1'=1) & (dado1'=0)+
                        attaccoC/(attaccoA+attaccoB+attaccoC)   :
(c'= max(c-dado1, 0)) & (turnoSq1'=1) & (dado1'=0);
//lancio del giocatore B
[] !terminato & turnoSq1=1 & y>0 & dado1=0->
                                    ((k>=1)?1:0)/k : (dado1'=1) +
                                    ((k>=2)?1:0)/k : (dado1'=2) +
                                    ((k>=3)?1:0)/k : (dado1'=3) +
                                    ((k>=4)?1:0)/k : (dado1'=4) +
                                    ((k>=5)?1:0)/k : (dado1'=5) +
                                    ((k>=6)?1:0)/k : (dado1'=6) +
                                    ((k>=7)?1:0)/k : (dado1'=7) +
                                    ((k>=8)?1:0)/k : (dado1'=8) +
                                    ((k>=9)?1:0)/k : (dado1'=9) +
                                    ((k>=10)?1:0)/k : (dado1'=10);
//se il giocatore B ha perso passa il turno al giocatore C
[] !terminato & turnoSq1=1 & dado1=0 & y<=0-> (turnoSq1'=2);
//con probabilità 1/3 attacca uno dei tre della squadra1 (se hanno
gli stessi punti)
[]!terminato & turnoSq1=1 & dado1!=0 & tuttiUguali1->
                        1/3 : (a'= max(a-dado1, 0)) & (turnoSq1'=2)
& (dado1'=0)+
                        1/3 : (b'= max(b-dado1, 0)) & (turnoSq1'=2)
& (dado1'=0)+
                        1/3 : (c'= max(c-dado1, 0)) & (turnoSq1'=2)
& (dado1'=0);
//se non hanno gli stessi punti usa la strategia 2
[]!terminato & turnoSq1=1 & dado1!=0 & !tuttiUguali1->
                        attaccoA/(attaccoA+attaccoB+attaccoC) : (a'=
max(a-dado1, 0)) & (turnoSq1'=2) & (dado1'=0)+
                        attaccoB/(attaccoA+attaccoB+attaccoC)  :
(b'= max(b-dado1, 0)) & (turnoSq1'=2) & (dado1'=0)+
                        attaccoC/(attaccoA+attaccoB+attaccoC)  :
(c'= max(c-dado1, 0)) & (turnoSq1'=2) & (dado1'=0);
//lancio del giocatore C
[] !terminato & turnoSq1=2 & z>0 & dado1=0->
                              ((k>=1)?1:0)/k : (dado1'=1) +
```

```
                                ((k>=2)?1:0)/k : (dado1'=2) +
                                ((k>=3)?1:0)/k : (dado1'=3) +
                                ((k>=4)?1:0)/k : (dado1'=4) +
                                ((k>=5)?1:0)/k : (dado1'=5) +
                                ((k>=6)?1:0)/k : (dado1'=6) +
                                ((k>=7)?1:0)/k : (dado1'=7) +
                                ((k>=8)?1:0)/k : (dado1'=8) +
                                ((k>=9)?1:0)/k : (dado1'=9) +
                                ((k>=10)?1:0)/k : (dado1'=10);
//se il giocatore Z ha perso risetta il turno a 0
[fineTurno1] !terminato & turnoSq1=2 & dado1=0 & z<=0->
(turnoSq1'=0) ;
//con probabilità 1/3 attacca uno dei tre della squadra1 (se hanno
gli stessi punti)
[fineTurno1]!terminato & turnoSq1=2 & dado1>0 & tuttiUguali1->
                        1/3 : (a'= max(a-dado1, 0)) & (turnoSq1'=0)
& (dado1'=0)+
                        1/3 : (b'= max(b-dado1, 0)) & (turnoSq1'=0)
& (dado1'=0)+
                        1/3 : (c'= max(c-dado1, 0)) & (turnoSq1'=0)
& (dado1'=0);
//se non hanno gli stessi punti usa la strategia 2
[fineTurno1]!terminato & turnoSq1=2 & dado1>0 & !tuttiUguali1->
                        attaccoA/(attaccoA+attaccoB+attaccoC)  :
(a'= max(a-dado1, 0)) & (turnoSq1'=0) & (dado1'=0)+
                        attaccoB/(attaccoA+attaccoB+attaccoC)  :
(b'= max(b-dado1, 0)) & (turnoSq1'=0) & (dado1'=0)+
                        attaccoC/(attaccoA+attaccoB+attaccoC)  :
(c'= max(c-dado1, 0)) & (turnoSq1'=0) & (dado1'=0);
endmodule
module MSquadra2 //modulo con strategia 2 (si attacca un giocatore
finché non viene sconfitto)
turnoSq2: [0..2] init 0;
dado2 : [0..10] init 0;
//variabili dei giocatori della squadra1
x: [0..10] init 10;
y: [0..10] init 10;
z: [0..10] init 10;
//lancio del giocatore A
```

```
[turno2] !terminato & turnoSq2=0 & a>0->
                                        ((k>=1)?1:0)/k : (dado2'=1) +
                                        ((k>=2)?1:0)/k : (dado2'=2) +
                                        ((k>=3)?1:0)/k : (dado2'=3) +
                                        ((k>=4)?1:0)/k : (dado2'=4) +
                                        ((k>=5)?1:0)/k : (dado2'=5) +
                                        ((k>=6)?1:0)/k : (dado2'=6) +
                                        ((k>=7)?1:0)/k : (dado2'=7) +
                                       ((k>=8)?1:0)/k : (dado2'=8) +
                                        ((k>=9)?1:0)/k : (dado2'=9) +
                                      ((k>=10)?1:0)/k : (dado2'=10);
//se il giocatore A ha perso passa il turno al giocatore B
[turno2] !terminato & turnoSq2=0 & dado2=0 & a<=0-> (turnoSq2'=1);
//con probabilità 1/3 attacca uno dei tre della squadra1 (se hanno
gli stessi punti)
[]!terminato & turnoSq2=0 & dado2!=0 & tuttiUguali2 & !anomalo2->
                      xVivo/numVivi : (x'= max(x-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0)+
                      yVivo/numVivi : (y'= max(y-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0)+
                      zVivo/numVivi : (z'= max(z-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0);
//se non hanno gli stessi punti usa la strategia 2
[]!terminato & turnoSq2=0 & dado2!=0 & !tuttiUguali2 & !anomalo2->
                      PerdenteX/numVivi : (x'= max(x-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0)+
                      PerdenteY/numVivi : (y'= max(y-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0)+
                      PerdenteZ/numVivi : (z'= max(z-dado2, 0)) &
(turnoSq2'=1) & (dado2'=0);
[] !terminato & turnoSq2=1 & b>0 & dado2=0->
                           ((k>=1)?1:0)/k : (dado2'=1) +
                           ((k>=2)?1:0)/k : (dado2'=2) +
                           ((k>=3)?1:0)/k : (dado2'=3) +
                           ((k>=4)?1:0)/k : (dado2'=4) +
                           ((k>=5)?1:0)/k : (dado2'=5) +
                           ((k>=6)?1:0)/k : (dado2'=6) +
                           ((k>=7)?1:0)/k : (dado2'=7) +
                           ((k>=8)?1:0)/k : (dado2'=8) +
```

```
                              ((k>=9)?1:0)/k : (dado2'=9) +
                              ((k>10)?1:0)/k : (dado2'=10);
//se il giocatore B ha perso passa il turno al giocatore C
[] !terminato & turnoSq2=1 & dado2=0 & b<=0-> (turnoSq2'=2);
//con probabilità 1/3 attacca uno dei tre della squadra1 (se hanno
gli stessi punti)
[]!terminato & turnoSq2=1 & dado2!=0 & tuttiUguali2 & !anomalo2->
                        xVivo/numVivi : (x'= max(x-dado2, 0)) &
(turnoSq2'=2) & (dado2'=0)+
                        yVivo/numVivi : (y'= max(y-dado2, 0)) &
(turnoSq2'=2) & (dado2'=0)+
                        zVivo/numVivi : (z'= max(z-dado2, 0)) &
(turnoSq2'=2) & (dado2'=0);
//se non hanno gli stessi punti usa la strategia 2
[]!terminato & turnoSq2=1 & dado2!=0 & !tuttiUguali2 & !anomalo2->
                        PerdenteX/numVivi : (x'= max(x-dado2, 0)) &
(turnoSq2'=2) & (dado2'=0)+
                        PerdenteY/numVivi : (y'= max(y-dado2, 0)) &
(turnoSq2'=2) & (dado2'=0)+
                        PerdenteZ/numVivi : (z'= max(z-dado2, 0)) &
(turnoSq2'=2) & (dado2'=0);
//lancio del giocatore C
[] !terminato & turnoSq2=2 & c>0 & dado2=0->
                                  ((k>=1)?1:0)/k : (dado2'=1) +
                                  ((k>=2)?1:0)/k : (dado2'=2) +
                                  ((k>=3)?1:0)/k : (dado2'=3) +
                                  ((k>=4)?1:0)/k : (dado2'=4) +
                                  ((k>=5)?1:0)/k : (dado2'=5) +
                                  ((k>=6)?1:0)/k : (dado2'=6) +
                                  ((k>=7)?1:0)/k : (dado2'=7) +
                                  ((k>=8)?1:0)/k : (dado2'=8) +
                                  ((k>=9)?1:0)/k : (dado2'=9) +
                                  ((k>=10)?1:0)/k : (dado2'=10);
//se il giocatore Z ha perso risetta il turno a 0
[fineTurno2] !terminato & turnoSq2=2 & dado2=0 & c<=0->
(turnoSq2'=0) ;
//con probabilità 1/3 attacca uno dei tre della squadra1 (se hanno
gli stessi punti)
[fineTurno2]!terminato & turnoSq2=2 & dado2>0 & tuttiUguali2 & !
```

```
anomalo2->
                        xVivo/numVivi : (x'= max(x-dado2, 0)) &
(turnoSq2'=0) & (dado2'=0)+
                        yVivo/numVivi : (y'= max(y-dado2, 0)) &
(turnoSq2'=0) & (dado2'=0)+
                        zVivo/numVivi : (z'= max(z-dado2, 0)) &
(turnoSq2'=0) & (dado2'=0);
//se non hanno gli stessi punti usa la strategia 2
[fineTurno2]!terminato & turnoSq2=2 & dado2>0 & !tuttiUguali2 & !
anomalo2 ->
                        PerdenteX/numVivi : (x'= max(x-dado2, 0)) &
(turnoSq2'=0) & (dado2'=0)+
                        PerdenteY/numVivi : (y'= max(y-dado2, 0)) &
(turnoSq2'=0) & (dado2'=0)+
                        PerdenteZ/numVivi : (z'= max(z-dado2, 0)) &
(turnoSq2'=0) & (dado2'=0);
endmodule
rewards
      [turno1] true: 1;
      [turno2] true: 1;
endrewards
```

Appendice C: Model.jj

```
options {
}
PARSER_BEGIN(Model)
public class Model {
  public static void main(String args[]) {
    Model parser;
    if (args.length == 0) {
      System.out.println("GameDSL Parser:  Reading from standard
input . . .");
      parser = new Model(System.in);
    } else if (args.length == 1) {
      System.out.println("GameDSL Parser:  Reading from file " +
args[0] + " . . .");
      try {
        parser = new Model(new java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
        System.out.println("GameDSL Parser:  File " + args[0] + "
not found.");
        return;
      }
    } else {
      System.out.println("GameDSL Parser:  Usage is one of:");
      System.out.println("         java GameDSLParser <
inputfile");
      System.out.println("OR");
      System.out.println("         java GameDSLParser inputfile");
      return;
    }
    try {
      parser.Model();
      System.out.println("GameDSL Parser:  GameDSL program parsed
successfully.");
    } catch (ParseException e) {
      System.out.println("GameDSL Parser:  Encountered errors
```

```
during parse.");
      e.printStackTrace();
    }
  }
}
PARSER_END(Model)

/* WHITE SPACE */
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
 /* COMMENTS */
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])*
"*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"]
(~["*"])* "*"))* "/">
}
/*NOT TERMINAL*/
TOKEN:{
      < INIT: "init" >
      | < CLASS: "class" >
      | < TERMINATION: "termination" >
      | < STEP: "step" >
      | < IF: "if" >
      | < ELSE: "else">
      | < INTEGER: "int" >
      | < BOOLEAN: "bool" >
      | < DOUBLE: "double" >
      | < LIST: "list" >
      | < DISTRIBUTION: "distribution" >
      | < GRAPH: "graph" >
```

```
      | < EDGES: "edges" >
      | < VOID: "void" >
      | < RANDOM: "random" >
      | < PICK: "pick" >
      | < FILTER: "filter" >
      | < TRUE: "true" >
      | < FALSE: "false" >
      | < EXISTS: "exists" >
      | < FORALL: "forall" >
      | < RETURN: "return" >
}
/*NUMBER*/
TOKEN:{
       < INT_LIT: (["0"-"9"])+>
}
/*CHAR*/
TOKEN:{
       < ID: (["a" - "z"] | ["A" - "Z"] | "_" | ["0" – "9"])+>
}

void Model():
{}
{
(ClassDef() | VarDef() | DistDef())* ["init" "{" Com() "}"]
"termination" "{" Expr() "}" "step" "{" (Com())* "}"
<EOF>
}
void Expr():
{}
{
AndOrExp() ["?" Expr() ":" Expr()]
}
void AndOrExp():
{}
{
BExpr() ("&" BExpr() | "|" Bexpr())*
}
void Bexpr():
{}
```

```
{
AritExp()[(">" | "<" | "==" | "!=")AritExp()]
}
void AritExp():
{}
{
Term() ("+" Term() | "-" Term())*
}
void Term():
{}
{
Unari() ( "*" Unari() | "/" Unari())*
}
void Unari():
{}
{
Primary() |"!" Primary() | "-" Primary()
}
 void Primary():
{}
{
<INT_LIT> | "(" [Expr()] ")" | "true" | "false"  | "random"
"("<INT_LIT> "," <INT_LIT>")"
| Pick()["."Path()] | "filter" "(" BasicType() <ID> "," Expr() ")"
"{"Expr()"}"["."<ID>] | "forall" "(" BasicType() <ID> "," Expr()
")" "{"Expr()"}" | "exists" "(" BasicType() <ID> "," Expr() ")"
"{"Expr()"}" | Path()
}
void Pick():
{}
{
"pick" "("Expr()"," Expr() ")"
}
 void BasicType():
{}
{
"int" | "bool" | "double" | <ID>
}
void CompoundType():
```

```
{}
{
"list" BasicType() | BasicType()
}
void Tuple():
{}
{
"(" [Expr() ("," Expr())*] ")"
}
void Com():
{}
{
LOOKAHEAD(2) CompoundType() <ID> "=" Expr() ";" |
[Pick()"."]Path() ["=" Expr()]";"
| "if" "(" Expr() ")" "{" (Com())*"}" [LOOKAHEAD(2) "else" "{"
(Com())*"}"]
| "return" [Expr()]";"
}
void PathBase():
{}
{
<ID> [ "("[Expr() ("," Expr())*] ")" ] [ "[" Expr() "]" ]
}
void Path():
{}
{
PathBase()["."Path()]
}
void MethodDef():
{}
{
(CompoundType() | "void") <ID> "(" [BasicType() <ID> ("," 
BasicType() <ID>)*]")" "{" (Com())*  "}"
}
void DistDef():
{}
{
"distribution" <ID> "("BasicType() <ID> "," <ID>")""{"Expr()"}"
}
```

```
void ClassDef():
{}
{
"class" <ID> "{" (LOOKAHEAD(3) VarDef() | MethodDef())*
["init""("[BasicType() <ID> ("," BasicType() <ID>)*]")"
"{"(Com())*"}"] "}"
}
void VarDef():
{}
{
BasicType() <ID> ["=" Expr()] ";"
| "list" BasicType() <ID> "=" "{" [Tuple() ("," Tuple())*]"}" ";"
| "graph" BasicType() <ID> "=" "{"[<ID>(","<ID>)*]"}" ";"
| "edges" <ID> "=" "{" "("<INT_LIT> "," <INT_LIT> ")" ("," "("
<INT_LIT> "," <INT_LIT> ")")* "}" ";"
}
```