UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

VERIFICA PARAMETRICA DI PROTOCOLLI FAULT-TOLERANT

Relatore: Prof. Silvio GHILARDI
Correlatore: Dr. Ing. Silvio RANISE
Prof.ssa Elena PAGANI

Tesi di Laurea di:
Francesco ALBERTI
Matr. Nr. 733774

ANNO ACCADEMICO 2008/2009

Indice

1 Introduzione

Verificare la correttezza di un sistema informatico è un problema complesso e ancora oggi oggetto di ricerca. Le due più diffuse strategie di verifica, adottate per il controllo della stragrande maggioranza dei sistemi software, sono la *simulazione* e il *testing*. Anche se queste due tecniche sono molto efficaci nelle fasi iniziali di verifica, quando il sistema presenta solitamente molte imprecisioni e difetti, la loro effettiva capacità di trovare errori diminuisce drasticamente mano a mano che il software viene corretto: a questo punto le due tecnologie citate necessitano un'enormità di risorse (soprattutto in termini di tempo) per riuscire a scovare bachi più "nascosti". Inoltre, una volta che queste due tecniche non rilevano più errori nel sistema non si può sapere se quest'ultimo sia effettivamente corretto oppure se presenti ancora errori non rintracciabili con la simulazione o col testing. Proprio per questi motivi, le due tecniche di verifica citate sono sempre più inefficaci quanto più è alta la complessità del sistema da controllare.

Un'alternativa valida alla simulazione e al testing è rappresentata dai metodi formali. Mentre le prime due tecniche esplorano solo una parte dei comportamenti possibili del software, lasciando aperta la possibilità che inesplorate esecuzioni presentino errori fatali per la correttezza dello stesso, con la verifica formale viene eseguita un'esplorazione esaustiva di tutti i possibili comportamenti.

All'interno dei metodi formali, il model-checking si è imposto all'attenzione sia del mondo accademico che del mondo dell'industria per alcune sue

caratteristiche peculiari e particolarmente attraenti. In primo luogo, il model-checking è una tecnica completamente automatica che non richiede interazioni complesse con l'utente, dopo la fase preliminare di modellizzazione del sistema. In secondo luogo, le tecniche di model-checking, qualora rilevino che il sistema non soddisfi le caratteristiche specificate, sono in grado di produrre controesempi consistenti in tracce di esecuzione che testimoniano il comportamento indesiderato.

I primi algoritmi di model checking a stati finiti operavano su particolari sistemi di transizione chiamati *strutture di Kripke* che venivano utilizzati per rappresentare esplicitamente tutte le possibili esecuzioni del sistema. Questa rappresentazione era funzionale per sistemi concorrenti con un pochi processi, in quanto il numero degli stati rimaneva sufficientemente piccolo da poter essere trattato con tecniche esplicite, ma si dimostrò inefficiente per verificare sistemi complessi nei quali interagivano concorrentemente molte componenti.

La rappresentazione implicita degli stati proposta da McMillan nel 1987 [12, 30] permise di verificare sistemi sempre più complessi. Combinando questa nuova forma di rappresentazione simbolica (basata su una struttura dati particolarmente efficiente per trattare formule booleane, le *OBDD - Ordered Binary Decision Diagrams*) con algoritmi proposti da Emerson e Clarke [14] all'inizio degli anni '80, divenne possibile verificare sistemi con più di 10^{20} stati [15], e successivamente questo numero aumentò ancora di più fino ad arrivare [10, 11] a 10^{120} stati.

Al giorno d'oggi le tecniche di model checking a stati finiti sono state implementate in molti sistemi, ad esempio NuSMV [1] e Spin [2]; esse hanno avuto - e continuano ad avere - molto successo in ambito sia accademico che industriale, tuttavia presentano un punto debole: sebbene il numero degli stati possibili della rappresentazione formale del sistema possa essere molto grande, deve essere sempre un numero *finito*. Ciò implica che le tecniche di model checking a stati finiti

- non permettono di eseguire una verifica *parametrica* nel numero dei processi di un sistema software: in questo caso infatti il numero di processi concorrenti è sì finito, ma *ignoto*!

- non permettono di verificare il comportamento di processi che manipo-

lano dati definiti su domini illimitati (interi, reali, ecc...)

Con l'avvento dei sistemi distribuiti, sono stati sviluppati protocolli cosiddetti *parametrici*, ovvero progettati in modo da garantire il corretto funzionamento del sistema indipendentemente dal numero di processi da cui sono composti. Una parte di questi protocolli viene utilizzata nei sistemi *fault-tolerant*, per permettere a questi ultimi di continuare ad operare correttamente nonostante si verifichino guasti parziali al loro interno, ovvero guasti che influenzano il regolare funzionamento di alcune componenti, lasciandone allo stesso tempo completamente indifferenti altre [31].

Nel presente lavoro di tesi verrà presa in considerazione la verifica di questa classe di protocolli; l'idea è quella di verificare proprietà di *safety* (ovvero, intuitivamente, quella classe di proprietà che assicura che "niente di catastrofico può accadere nel sistema") dei sistemi nei quali vengono eseguiti questi protocolli indipendentemente dal numero dei processi che li costituiscono, ovvero eseguire una verifica parametrica. Questo tipo di verifica è molto difficile sia dal punto di vista teorico (in generale, infatti, è indecidibile) sia dal punto di vista pratico (poiché bisogna rappresentare in maniera compatta un insieme infinito di stati del sistema, visto che non è conosciuto il numero di processi che lo compongono). Dal punto di vista formale si utilizzerà un framework logico recentemente sviluppato [20] che permette di specificare sistemi distribuiti (sia la topologia del sistema stesso che le strutture dati manipolate dal sistema) in maniera dichiarativa.

Il lavoro di tesi prevede innanzitutto l'identificazione di un formalismo ad alto livello, derivato da [20], che permetta di esprimere in maniera astratta e precisa un'ampia classe di protocolli fault-tolerant. Verrà definito un linguaggio formale per la definizione di configurazioni iniziali e unsafe del sistema, transizioni, invarianti, assiomi e sarà sviluppato un traduttore per realizzare automaticamente file accettati dal tool MCMT, che offre un'implementazione del formalismo e dei risultati riportati in [20]. In secondo luogo, oltre alla parte relativa alla definizione del linguaggio ed allo sviluppo del traduttore, in questa tesi vi sarà anche una componente sperimentale che mira alla verifica parametrica di protocolli fault-tolerant mai verificati (parametricamente) prima. In questa seconda parte i protocolli verranno descritti formalmente

secondo il linguaggio definito nella prima parte della tesi e successivamente verificati utilizzando il tool MCMT.

1.1 Organizzazione della tesi

La tesi è divisa in due parti; la prima parte è composta dai capitoli 2, 3 e 4 mentre la seconda dai restanti capitoli 5, 6 e 7.

Nel Capitolo 2 viene descritto il funzionamento del tool MCMT: l'algoritmo di raggiungibilità a ritroso, i sistemi di transizione a variabili array, il modello *stopping failure* (che è adottato da MCMT per riuscire a gestire le guardie universali) e vengono riportati alcuni risultati teorici che stanno alla base del funzionamento del tool stesso.

Il Capitolo 3 è dedicato al linguaggio ad alto livello ideato per formalizzare il sistema (oppure il protocollo, una funzione scritta secondo qualche linguaggio di programmazione, ecc...) oggetto della verifica: verrà descritto brevemente il linguaggio sviluppato e le varie fasi di analisi implementate nel traduttore che può essere utilizzato per generare automaticamente file di specifica scritti secondo la sintassi accettata da MCMT.

Il Capitolo 4 introduce i sistemi *fault-tolerant* ed il problema del *Reliable Broadcast*.

La seconda parte della tesi riporta un caso di studio: la verifica di protocolli *fault-tolerant* (tratti da [13]) utilizzando MCMT.

Nel Capitolo 5 viene studiato il problema del *Reliable Broadcast* in un sistema che ammette solo fallimenti *crash*. Il capitolo successivo è dedicato all'analisi dello stesso protocollo assumendo fallimenti *send-omission*: seguento la via tracciata dagli autori di [13] verrà verificato che il protocollo del Capitolo 5 non riesce a risolvere il problema del *Reliable Broadcast* in caso di fallimenti in *send-omission*, ed inoltre che la prima modifica proposta dagli autori non è sufficiente a risolvere il problema (come dimostrato su [13]).

Infine, verrà effettuata una seconda modifica al protocollo (sempre seguendo le indicazioni riportate su [13]) e verrà verificato che questa nuova versione riesce a risolvere il problema del *Reliable Broadcast* se il modello di fallimento assunto ammette errori durante l'invio dei messaggi.

Parte I

2

MCMT - Model Checking Modulo Theories

In [20] viene presentato un approccio completamente dichiarativo alla verifica di proprietà di *safety* di sistemi a stati infiniti le cui variabili sono array tramite un algoritmo di *raggiungibilità a ritroso* (*backward reachability*). Questi sistemi possono essere utilizzati come astrazione di protocolli parametrici, programmi sequenziali che operano su array, ecc... L'idea di base è quella di utilizzare classi di formule quantificate del primo ordine per rappresentare un insieme infinito di stati del sistema in modo che il calcolo della preimmagine si riconduce ad una manipolazione simbolica di queste formule. Utilizzando apposite teorie sugli elementi e sugli indici degli array, si possono specificare dichiarativamente sia i dati sui quali opera il sistema e la sua topologia (nel caso di sistemi parametrici) o proprietà degli indici degli array (nel caso di programmi scritti secondo un paradigma di programmazione imperativo).

Nell'infrastruttura descritta in [20], il punto chiave per meccanizzare l'algoritmo di raggiungibilità a ritroso è ridurre il test di punto fisso e la verifica di proprietà di *safety* del sistema ad un problema SMT (*Soddisfacibilità Modulo Teorie*) di formule del primo ordine contenenti quantificatori (universali). Sotto opportune ipotesi sulle teorie degli indici e degli elementi degli array, questi problemi sono decidibili [20] integrando un modulo per l'istanziazione dei quantificatori a tecniche per risolvere i problemi SMT costituiti da formule senza quantificatori. In [21], [24] vengono descritte euristiche per ridurre il numero delle variabili quantificate e – ancor più importante – di istanze preservando la completezza della risoluzione SMT. Sfortunatamente,

la decidibilità del test di punto fisso e della verifica di proprietà di *safety* non è sufficiente per garantire la terminazione dell'algoritmo di *backward reachability*. Teoricamente, la terminazione per questa procedura è garantita per sistemi *ben strutturati* (*well-structured systems*) [3]. In [20] viene indicato come adattare i risultati presenti in letteratura al framework utilizzato per MCMT.

2.1 Il funzionamento di MCMT

Prima di descrivere l'approccio al model checking di sistemi a stati infiniti implementato da MCMT è conveniente richiamare altri due approcci distinti e complementari tra tutti quelli disponibili in letteratura.

Il primo approccio [3] è basato sulla nozione di sistema ben strutturato (*well-structured system*), e recentemente è stato implementato in due sistemi (si veda, ad esempio, [4, 5]) capaci di verificare automaticamente diversi protocolli di mutua esclusione e cache coherence. Il successo di questi tool è dovuto in buona parte alla loro capacità di calcolare accurati test di punto fisso in modo da ridurre il numero di iterazioni della procedura di *backward reachability*. Il test di punto fisso è implementato "inglobando" una configurazione vecchia (cioè una rappresentazione finita di un insieme potenzialmente infinito di stati) in una nuova pre-immagine; se la nuova pre-immagine è considerata "ridondante" (cioè che non fornisce alcuna nuova informazione riguardo l'insieme degli stati raggiungibili) può essere scartata senza perdita di precisione. Un problema aggiuntivo è che i vincoli sono utilizzati solamente per rappresentare i dati manipolati dal sistema, mentre la sua topologia viene codificata da strutture dati apposite. Ciò richiede un'implementazione specifica di algoritmi sia per calcolare pre-immagini e configurazioni da inglobare in esse ogni volta che la topologia del sistema da verificare viene modificata. Al contrario, MCMT utilizza particolari classi di formule del primo ordine per rappresentare configurazioni parametriche rispetto alla teoria dei dati e a quella della topologia del sistema, in modo che il calcolo della pre-immagine si riduce ad un numero fisso di manipolazioni logiche e il test di punto fisso è ricondotto alla risoluzione di problemi SMT contenenti variabili universalmente quantificate. Per meccanizzare questi test, viene utilizzata

una procedura per istanziare i quantificatori (questa procedura corrisponde all'inglobamento delle configurazioni nelle nuove pre-immagini descritte precedentemente). È interessante notare come questa nozione di "inglobamento" può essere ritrovata [20] nel framework logico su cui si basa MCMT, fatto che permette di importare in questo ambito i risultati teorici di decidibilità esposti in [3] riguardanti la *backward reachability*. Un altro vantaggio dell'approccio dichiarativo (sul quale si basa MCMT) rispetto a quello proposto in [3] è l'applicabilità di MCMT in ambiti molto più eterogenei rispetto a quelli di [4, 5] (MCMT è stato utilizzato per verificare programmi sequenziali - come algoritmi di ordinamento - che esulano dalle possibilità dei sistemi descritti in [4, 5]): l'utilizzo delle teorie per specificare i dati e la topologia permette di modellare diverse ed eterogenee classi di sistemi in un modo del tutto naturale. Per di più, anche se la procedura deputata all'istanziazione dei quantificatori diventa incompleta con teorie sempre più ricche, può comunque essere utilizzata e poterebbe permettere in ogni caso la verifica di proprietà di *safety* del sistema.

Il secondo approccio al model checking di sistemi a stati infiniti è basato sull'astrazione dei predicati (*predicate abstraction*), proposta inizialmente in [25]. L'idea alla base di questo approccio è astrarre il sistema (potenzialmente a stati infiniti) per il quale si vuole verificare una certa proprietà di safety, ad uno a stati finiti, per poter applicare le tecniche di model checking a stati finiti, raffinando eventuali tracce spurie (se ve ne sono) utilizzando procedure di decisione o un SMT solver. Questa tecnica è stata implementata in diversi tools, ed è spesso combinata con algoritmi di interpolazione per la fase di raffinamento. Come viene puntualizzato in [17, 29], l'astrazione dei predicati deve essere adattata con attenzione quando la quantificazione (universale) viene utilizzata per specificare le transizioni del sistema o le sue proprietà, e questo è il caso per i problemi affrontati da MCMT. Sorgono due problemi da risolvere. Il primo è trovare un insieme appropriato di predicati per calcolare l'astrazione del sistema. Di fatto, oltre alle variabili di sistema, potrebbero esserci variabili universalmente quantificate. Il secondo problema è che il calcolo dell'astrazione ed il suo raffinamento richiede di risolvere delle *proof obligation* contenenti quantificatori universali. Quindi c'è bisogno di un'adatta istanziazione dei quantificatori per poter utilizzare le procedure di

decisione o un SMT solver. Il primo problema viene risolto tramite la Skolemizzazione [17] o fissando il numero delle variabili del sistema [29] tale che possono essere utilizzate le tecniche standard di astrazione dei predicati. Il secondo problema è risolto utilizzando una semplice e incompleta procedura di istanziazione dei quantificatori. Se da un lato le tecniche di istanziazione dei quantificatori sono economiche in termini computazionali e facili da implementare, le euristiche utilizzate per istanziare i quantificatori sono largamente imprecise e non permettono il rilevamento delle ridondanze dovute alla permutazione delle variabili, simmetrie interne, ecc... Esperimenti eseguiti con MCMT imitando queste semplici strategie di instanziazione, hanno fatto registrare performance scadenti. Per questi motivi è lecito pensare che il successo dell'astrazione dei predicati utilizzata in [17, 29] è dovuto ad euristiche ben ingegnerizzate che permettono la ricerca ed il raffinamento dell'insieme dei predicati per astrarre il sistema. L'implementazione attuale di MCMT è ortogonale rispetto all'approccio d'astrazione dei predicati; infatti è caratterizzato da una vasta procedura di istanziazione dei quantificatori (che è completa per alcune teorie sugli indici ed è migliorata da euristiche - che ne preservano la completezza - per evitare istanziazioni inutili), ma esegue solo una forma primitiva di astrazione dei predicati, chiamata *astrazione della segnatura* (*signature abstraction*). Un'altra grossa differenza è la modalità con la quale viene utilizzata la tecnica di astrazione in MCMT: l'insieme degli stati raggiungibili all'indietro è calcolato precisamente, mentre le tecniche di astrazione vengono sfruttate solo per cercare gli invarianti, che - una volta verificata la loro natura effettiva di invarianti - saranno utilizzati per potare l'albero di ricerca degli stati raggiungibili. Dal momento che insiemi di stati vengono rappresentati da formule, trovare ed utilizzare gli invarianti è estremamente semplice, pertanto le modalità per aiutare a risolvere la tensione tra model checking e tecniche deduttive sono state discusse a lungo in letteratura, e sono tutt'ora problematiche per i tool descritti in [4, 5] dove l'insieme degli stati sono rappresentati tramite strutture dati apposite.

2.2 Notazione

Un'introduzione generale sui sistemi *array-based* è riportata in [20],[24], [21]. In questa tesi verranno utilizzate notazioni e definizioni introdotte nel secondo paragrafo di [21].

Si assumono le comuni nozioni sintattiche (e.g., segnatura, variabile, termine, atomo, letterale e formula) e semantiche (e.g., struttura, sottostruttura, verità, soddisfacibilità, e validità) della logica del primo ordine (si veda, ad esempio, [16]). Il simbolo d'uguaglianza $=$ è incluso in tutte le segnature considerate da qui in avanti. Una segnatura è *relazionale* se non contiene alcun simbolo di funzione mentre è *quasi-relazionale* se i suoi simboli di funzione sono tutti costanti (inviduali). Un'*espressione* è un termine, una formula atomica (cioè una formula ottenuta applicando un simbolo di predicato a termini), un letterale o una formula. Sia $\underline{x}$ una tupla finita di variabili e Σ una segnatura; una $\Sigma(\underline{x})$-espressione è un'espressione costituita da simboli appartenenti a Σ dove al più sono libere le variabili presenti in $\underline{x}$.

Una *teoria* T (in accordo con [7]), è una coppia $(\Sigma, \mathcal{C})$ dove Σ è una segnatura e $\mathcal{C}$ è una classe di Σ-strutture; le strutture in $\mathcal{C}$ sono i *modelli* di T. Da questo punto, per tutta la tesi sia $T = (\Sigma, \mathcal{C})$.

Una Σ-formula ϕ è *T-soddisfacibile* se esiste una Σ-struttura $\mathcal{M}$ in $\mathcal{C}$ tale che ϕ è vera in $\mathcal{M}$ sotto un opportuno assegnamento delle variabili libere di ϕ; (in simboli, $\mathcal{M} \models \phi$); è *valida* (in simboli $T \models \varphi$) se la sua negazione è T-insoddisfacibile. Due formule φ_1 e φ_2 sono *T-equivalenti* se $\varphi_1 \leftrightarrow \varphi_2$ è *T-valida*. Il *problema di soddisfacibilità modulo (una) teoria* T (indicato con $SMT(T)$) punta a stabilire la T-soddisfacibilità di una Σ-formula che non presenta quantificatori.

Una teoria $T = (\Sigma, \mathcal{C})$ è detta *localmente finita* se e solo se Σ è finita e, per ogni insieme finito di variabili $\underline{x}$, esistono finiti $\Sigma(\underline{x})$-termini $t_1, \ldots, t_{k_{\underline{x}}}$ tali che per ogni altro $\Sigma(\underline{x})$-termine u, vale $T \models u = t_i$ (per qualche $i \in \{1, \ldots, k_{\underline{x}}\}$). I termini $t_1, \ldots, t_{k_{\underline{x}}}$ sono chiamati termini $\Sigma(\underline{x})$-rappresentanti; se sono effettivamente calcolabili partendo da $\underline{x}$ (e t_i è calcolabile dato u), allora T è detta essere *effettivamente localmente finita* (da qui in poi 'localmente finita', significherà 'effettivamente localmente finita'). Se Σ è relazionale o quasi-relazionale, qualsiasi Σ-teoria T è localmente finita.

Una T-*partizione* è un insieme finito $C_1(\underline{x}), \ldots, C_n(\underline{x})$ di formule senza quantificatori tali che

$$T \models \forall \underline{x} \bigvee_{i=1}^{n} C_i(\underline{x})$$

e

$$T \models \bigwedge_{i \neq j} \forall \underline{x} \neg (C_i(\underline{x}) \wedge C_j(\underline{x}))$$

Un'*estensione definitoria per casi* $T' = (\Sigma', \mathcal{C}')$ di una teoria $T = (\Sigma, \mathcal{C})$ è ottenibile da T applicando (un numero finito di volte) la seguente procedura:

(i) si prenda una T-partizione $C_1(\underline{x}), \ldots, C_n(\underline{x})$ ed alcuni Σ-termini $t_1(\underline{x}), \ldots, t_n(\underline{x})$;

(ii) sia $\Sigma' = \Sigma \cup \{F\}$, dove F è un "nuovo" simbolo di funzione (cioè $F \notin \Sigma$) la cui arietà è uguale alla lunghezza di $\underline{x}$;

(iii) sia $\mathcal{C}'$ la classe delle Σ'-strutture $\mathcal{M}$ il cui Σ-ridotto è un modello di T e tali che

$$\mathcal{M} \models \bigwedge_{i=1}^{n} \forall \underline{x} \, (C_i(\underline{x}) \rightarrow F(\underline{x}) = t(\underline{x}))$$

Un'estensione definitoria per casi T' di una teoria T contiene quindi un numero finito di simboli di funzione in più rispetto a T, chiamati *funzioni definite per casi*. Non è difficile portare un qualsiasi problema $SMT(T')$ in uno equivalente $SMT(T)$ (si veda [21] per i dettagli).

Da qui in poi, verrà utilizzata una logica del primo ordine multi-sorta. Tutte le notazioni introdotte fin qui possono essere adattate ad un'infrastruttura multi-sorta. In particolare, da qui in poi, saranno fisse:

(i) una teoria $T_I = (\Sigma_I, \mathcal{C}_I)$ per gli indici i cui simboli appartengono ad un'unica sorta detta `INDEX`;

(ii) una teoria $T_E = (\Sigma_E, \mathcal{C}_E)$ per i dati i cui simboli appartengono ad un'unica sorta detta `ELEM`;

La teoria $A_I^E = (\Sigma, \mathcal{C})$ degli array con con indici I ed elementi E è ottenuta come combinazione di T_I e T_E in questo modo: `INDEX`, `ELEM`, e `ARRAY`

sono le uniche sorte di A_I^E, la segnatura è $\Sigma := \Sigma_I \cup \Sigma_E \cup \{\_[\_]\}$ dove $\_[\_] : \texttt{ARRAY}, \texttt{INDEX} \longrightarrow \texttt{ELEM}$ (intuitivamente, $a[i]$ denota l'elemento salvato nell'array a in corrispondenza dell'indice i); una struttura con tre sorti $\mathcal{M} = (\texttt{INDEX}^{\mathcal{M}}, \texttt{ELEM}^{\mathcal{M}}, \texttt{ARRAY}^{\mathcal{M}}, \mathcal{I})$ appartiene a $\mathcal{C}$ se e solo se $\texttt{ARRAY}^{\mathcal{M}}$ è l'insieme delle funzioni (totali) da $\texttt{INDEX}^{\mathcal{M}}$ a $\texttt{ELEM}^{\mathcal{M}}$, il simbolo di funzione $\_[\_]$ è interpretato come l'applicazione di una funzione (e.g. $a[i]$ è inteso come l'applicazione della funzione a su input i) e $\mathcal{M}_I = (\texttt{INDEX}^{\mathcal{M}}, \mathcal{I}_{|\Sigma_I})$, $\mathcal{M}_E = (\texttt{ELEM}^{\mathcal{M}}, \mathcal{I}_{|\Sigma_E})$ sono rispettivamente modelli di T_I e T_E (dove $\mathcal{I}_{|\Sigma_X}$ è la restrizione dell'interpretazione $\mathcal{I}$ ai simboli in Σ_X per $X \in \{I, E\}$).

Convenzioni notazionali. Da qui in poi verranno usate le seguenti convenzioni notazionali: d, e sono simboli che rappresentano variabili di sorta `ELEM`, a rappresenta variabli di sorta `ARRAY` ed $i, j, k, z, \ldots$ rappresentano variabili di sorta `INDEX`.

Una variabile sottolineata è un'abbreviazione di una tupla di variabili di lunghezza non specificata e, se $\underline{i} := i_1, \ldots, i_n$, la notazione $a[\underline{i}]$ sta per la tupla di termini $a[i_1], \ldots, a[i_n]$. Eventuali espressioni del tipo $\phi(\underline{i}, \underline{e}), \psi(\underline{i}, \underline{e})$ denotano $(\Sigma_I \cup \Sigma_E)$-formule senza quantificatori nelle quali occorrono al più le variabili $\underline{i} \cup \underline{e}$. Inoltre, $\phi(\underline{i}, \underline{t}/\underline{e})$ (o semplicemente $\phi(\underline{i}, \underline{t})$) indica la sostituzione dei Σ-termini $\underline{t}$ alle variabili $\underline{e}$.

Perciò, per esempio, $\phi(\underline{i}, a[\underline{i}])$ denota la formula ottenuta sostituendo $\underline{e}$ con $a[\underline{i}]$ nella formula senza quantificatori $\phi(\underline{i}, \underline{e})$.

Formule speciali. Una $\forall^I$-formula è una formula della forma $\forall \underline{i}.\phi(\underline{i}, a[\underline{i}])$, ed una $\exists^I$-formula è una formula della forma $\exists \underline{i}.\phi(\underline{i}, a[\underline{i}])$.

2.2.1 Sistemi a variabili array

Un *sistema (di transizione) a variabili array* è una tripla $\mathcal{S} = (a, I, \tau)$ dove

(i) a una variabile di sorta `ARRAY` e rappresenta lo *stato* del sistema;[1]

[1]Per semplicità (e senza perdita di generalità), la definizione è limitata a sistemi con solo una variabile a di sorta `ARRAY`. Negli esempi concreti però utilizzeremo sistemi con più variabili `ARRAY`.

(ii) $I(a)$ è una $\Sigma(a)$-formula rappresentante l'insieme degli *stati iniziali* del sistema;

(iii) $\tau(a, a')$ è una $(\Sigma \cup \Sigma_D)(a, a')$-formula che rappresenta le *transizioni* del sistema, dove a' è una copia rinominata di a e Σ_D è un insieme finito di simboli di funzione definitori per casi che non appartengono a $\Sigma_I \cup \Sigma_E$.

Dato un sistema a variabili array $\mathcal{S} = (a, I, \tau)$ e una formula $U(a)$, il *problema di safety* (o meglio, una sua istanza) viene risolto se è possibile determinare se esiste un numero naturale n tale che la formula

$$I(a_0) \wedge \tau(a_0, a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge U(a_n) \tag{2.1}$$

è A_I^E-soddisfacibile. Se tale n non esiste $\mathcal{S}$ è *safe* (in riferimento ad U); altrimenti, è *unsafe* - la A_I^E-soddisfacibilità della (2.1) implica l'esistenza di un'esecuzione (composta da n passi) che porta il sistema da uno stato rappresentato da I ad uno stato rappresentato da U.

Un approccio generale per risolvere le istanze del problema di safety è basato sul calcolo degli insiemi degli stati raggiungibili a ritroso partendo da uno stato rappresentato da U.

Per un $n \geq 0$, l'n-esima *pre-immagine* di una formula $K(a)$ è

$$Pre^n(\tau, K) := \begin{cases} K & \text{se } n = 0 \\ Pre(\tau, Pre^{n-1}(\tau, K)) & \text{altrimenti} \end{cases}$$

dove

$$Pre(\tau, K) \;\; := \;\; \exists a'.(\tau(a, a') \wedge K(a')). \tag{2.2}$$

Dati $\mathcal{S} = (a, I, \tau)$ e $U(a)$, la formula $Pre^n(\tau, U)$ descrive l'insieme degli stati raggiungibili a ritroso in n passi (per $n \geq 0$) partendo da uno stato in $U(a)$. Durante l'n-esima iterazione del ciclo dell'*algoritmo di raggiungibilità a ritroso* riportato in Figura 2.1, la variabile B contiene la formula

$$BR^n(\tau, U) := \bigvee_{i=0}^{n} Pre^i(\tau, U)$$

```
function BReach(U)
1   P ⟵ U; B ⟵ ⊥;
2   while (P ∧ ¬B is A_I^E-sat.) do
3       if (I ∧ P is A_I^E-sat.)
                then return unsafe;
4       B ⟵ P ∨ B;
5       P ⟵ Pre(τ, P);
6   end
7   return (safe, B);
```

Figura 2.1: L'algoritmo di raggiungibilità a ritroso.

Questa formula rappresenta l'insieme degli stati che sono raggiungibili a ritroso partendo dagli stati rappresentati da U con al massimo n passi, mentre la variabile P contiene la formula $Pre^n(\tau, U)$. L'algoritmo BReach, mentre calcola iterativamente $BR^n(\tau, U)$, controlla se il sistema è unsafe (linea 3, che può essere letta come $I \wedge Pre^n(\tau, U)$ è A_I^E-soddisfacibile) oppure se è stato raggiunto un punto fisso (cf. linea 2, che può essere letta come $\neg(BR^n(\tau, U) \rightarrow BR^{n-1}(\tau, U))$ è A_I^E-insoddisfacibile o, equivalentemente, che $(BR^n(\tau, U) \rightarrow BR^{n-1}(\tau, U))$ è A_I^E-valida). Quando BReach indica che il sistema è *safe* (cf. linea 7) la variabile B contiene una formula che descrive l'insieme degli stati che sono raggiungibili a ritroso partendo da U, formula che rappresenta anche un punto fisso. Tuttavia, perché BReach (Figura 2.1) sia una vera procedura (eventualmente non terminante), è indispensabile che *siano effettivamente calcolabili la A_I^E-soddisfacibilità del test di safety (linea 3) e di quello di punto fisso (linea 2).* Per garantire questi fatti, sono necessarie alcune ipotesi.

Una classe di formule, la cui A_I^E-soddisfacibilità è decidibile sotto ragionevoli condizioni, è descritta dai cosidetti $\exists^{A,I}\forall^I$-enunciati, cioè da enunciati del tipo

$$\exists \underline{a}\ \exists \underline{i}\ \forall \underline{j}\ \psi(\underline{i}, \underline{j}, \underline{a}[\underline{i}], \underline{a}[\underline{j}])$$

per formule ψ che non presentano quantificatori al loro interno:

Teorema 2.2.1 ([20]). *La A_I^E-soddisfacibilità di $\exists^{A,I}\forall^I$-enunciati è decidibile se*

(i) T_I è localmente finita ed è chiusa sotto sottostrutture;

(ii) i problemi $SMT(T_I)$ e $SMT(T_E)$ sono decidibili.

Da qui in poi siano valide le condizioni (i) e (ii) appena enunciate. Ora il problema è quello di identificare un apposito formato per formule inziali, unsafe e di transizioni in modo che i test di soddisfacibilità richiesti dall'algoritmo riportato in Figura 2.1 riguardino $\exists^{A,I}\forall^I$-enunciati (in modo da poter sfruttare il risultato teorico riportato nel Teorema 2.2.1). In [21], per riuscire a soddisfare questi requisiti, è stata adottata la seguente soluzione:[2]

(i) le formule iniziali I sono $\forall^I$-*enunciati*;

(ii) le formule U che descrivono insiemi di stati sono $\exists^I$-*enunciati*;

(iii) le formule $\tau(a, a')$ che descrivono le transizioni devono essere in una forma *funzionale*, cioè devono essere *disgiunzioni di* formule della forma

$$\exists \underline{i}\, (\phi_L(\underline{i}, a[\underline{i}]) \wedge a' = \lambda j. F_G(\underline{i}, a[\underline{i}], j, a[j]) \tag{2.3}$$

dove ϕ_L è detta *guardia* (in [20] è chiamata *componente locale*), e F_G è una funzione definita per casi (chiamata *componente globale*).[3]

Le assunzioni (ii)-(iii) sono motivate dal fatto che è possibile mostrare [24] che *se K è una $\exists^I$-formula e τ è in forma funzionale, $Pre(\tau, K)$ è equivalente ad una $\exists^I$-formula.*

A questo punto è lecito chiedersi quanto siano restrittive le condizioni (i)-(ii)-(iii), cioè fino a che punto le applicazioni concrete che si vogliono descrivere le soddisfano. Fortunatamente i benchmark standard soddisfano

[2] In [20] viene identificato un formato più ampio che soddisfa questi requisiti, tuttavia questo formato non è stato implementato in MCMT.

[3] Va notato che le formule della forma (2.3) possono essere scritte come formule del primo ordine sostituendo l'equazione funzionale $a' = \lambda j. F_G(\underline{i}, a[\underline{i}], j, a[j])$ con $\forall j\, a'[j] = F_G(\underline{i}, a[\underline{i}], j, a[j])$). Per abuso di notazione viene sempre denotato da A_I^E una qualsiasi estensione definitoria per casi di A_I^E.

queste tre restrizioni, quindi questo framework teorico è *effettivamente* utilizzabile, tuttavia spesso le transizioni sono disgiunzioni di formule aventi il seguente formato più generale

$$\exists \underline{i}\,(\phi_L(\underline{i}, a[\underline{i}]) \wedge \forall k\,\psi(\underline{i}, k, a[\underline{i}], a[k]) \wedge a' = \lambda j.F_G(\underline{i}, a[\underline{i}], j, a[j])) \qquad (2.4)$$

In altre parole, formule tipo (2.4) (contrariamente alle formule di tipo (2.3)) contengono la sottoformula universale $\forall k\,\psi(\underline{i}, k, a[\underline{i}], a[k])$; queste transizioni sono dette *transizioni con guardie universali espresse in forma funzionale*. Ora la presenza di tali guardie universali modifica l'intero framework teorico , non è possibile dimostrare che il calcolo della preimmagine produca $\exists^I$-formule, e non è affatto evidente come possa venire in aiuto il Teorema 2.2.1.[4] Fortunatamente, tuttavia, guardie universali possono essere eliminate introducendo un modello di sistema a variabili array detto *stopping failures*; questo fatto è un punto chiave degli algoritmi di ricerca a ritroso descritti in [4], [5] e può essere ricatturata nello scenario dichiarativo sul quale si basa questo lavoro.

2.3 Loop principale: raggiungibilità a ritroso

L'algoritmo di *backward reachability* si basa sul calcolo delle formule $BR^n(\tau, U)$, incrementando n di volta in volta; per ogni nuovo $BR^n(\tau, U)$ calcolato viene eseguito il test di *safety* che consiste nel verificare la A_I^E-soddisfacibilità della formula $BR^n(\tau, U) \wedge I$. Se questa formula è A_I^E-soddisfacibile il sistema non è *safe*, in quanto verificare che $BR^n(\tau, U) \wedge I$ sia A_I^E-soddisfacibile significa trovare un cammino di lunghezza n che porta dallo stato iniziale alla configurazione unsafe.

L'algoritmo di calcolo delle preimmagini $BR^n(\tau, U)$ può essere interrotto se $\neg(BR^n(\tau, U) \rightarrow BR^{n-1}(\tau, U))$ è A_I^E-insoddisfacibile (tale verifica è nominata *test di punto fisso*), in quanto nessun nuovo stato potrà essere trovato. In questo caso, se il test di *safety* indica che $BR^n(\tau, U) \wedge I$ è insoddisfacibile si può dedurre che il sistema è *safe*.

[4] Si potrebbe comunque continuare ad utilizzare il Teorema 2.2.1 per la ricerca degli invarianti, si veda [9] dove Teorema 2.2.1 viene dimostrato (per il caso speciale nel quale T_I è la teoria di ordini lineari) ed è usato per questo scopo.

$$\frac{K \quad [K \text{ è primitiva differenziata}]}{\mathsf{Pre}(\tau_1, K) \mid \cdots \mid \mathsf{Pre}(\tau_m, K)}\ \mathsf{PreImg}$$

$$\frac{K}{K_1 \mid \cdots \mid K_n}\ \mathsf{Beta}$$

$$\frac{K \quad [K \text{ è } A_I^E\text{-insoddisfacibile}]}{\times}\ \mathsf{NotAppl}$$

$$\frac{K \quad [I \wedge K \text{ è } A_I^E\text{-soddisfacibile}]}{\mathsf{UnSafe}}\ \mathsf{Safety}$$

$$\frac{K \quad [K \wedge \bigwedge\{\neg K' | K' \preceq K\} \text{ è } A_I^E\text{-insoddisfacibile}]}{\times}\ \mathsf{FixPoint}$$

Figura 2.2: Regole di inferenza del calcolo implementato da MCMT

In Figura 2.2 viene introdotto un calcolo di tipo *Tableaux* utilizzato da MCMT per implementare l'algoritmo di raggiungibilità all'indietro [21]. Il tableau viene inizializzato con la $\exists^I$-formula $U(a)$ rappresentante l'insieme degli stati che non devono essere raggiunti dal sistema (cioè gli stati *unsafe*). Il calcolo della preimmagine è realizzato applicando la regola `PreImg` (vengono utilizzate le parentesi quadre per indicare la condizione di applicabilità della regola), dove $\mathtt{Pre}(\tau_h, K)$ calcola la $\exists^I$-formula che è logicamente equivalente a $Pre(\tau_h, K)$. Dal momento che le $\exists^I$-formule che etichettano le risultanti della regola `PreImg` potrebbero non essere primitive e differenziate (a causa delle espressioni `if-then-else` innestate e dell'incompletezza della distinzione delle variabili) bisogna applicare la regola `Beta` ad una $\exists^I$-formula in modo da eliminare condizionali tramite lo spezzamento dei casi e derivando $K_1, \ldots, K_n$ $\exists^I$-formule primitive e differenziate la cui disgiunzione è A_I^E-equivalente a K. Continuando ad applicare le regole `PreImg` e `Beta` è possibile costruire un albero i cui nodi sono etichettati con $\exists^I$-formule la cui disgiunzione è equivalente a $BR^n(\tau, U)$ per qualche $n \geq 0$. Infatti non è necessario espandere completamente l'albero; è inutile applicare la regola `PreImg` ad un nodo ν etichettato da una $\exists^I$-formula A_I^E-insoddisfacibile (regola `NotAppl`). Si può terminare l'intera ricerca grazie al test di *safety* (regola `Safety`), nel quale viene estratto da un ramo dell'albero di ricerca un'esecuzione - cioè una sequenza di transizioni - che porta da uno stato descritto dalla formula I ad uno stato rappresentabile con la formula U. Data

una formula che descrive una foglia dell'albero di ricerca, è possibile determinare se è possibile proseguire la ricerca da quella foglia utiilizzando la regola `FixPoint`: se si verifica che K implica (modulo A_I^E) la disgiunzione delle formule K' che etichettano i nodi che precedono K, l'esplorazione dell'albero sotto K viene bloccata perché ridondante.

Per applicare le regole Safety e FixPoint è necessario verificare la A_I^E-soddisfacibilità di $\exists^I\forall^I$-formule, cioè formule contenenti un'alternanza di quantificatori sulle variabili di sorta Ind. Come spiegato sopra, nella Sezione 2.1, MCMT integra una procedura di istanziazione dei quantificatori con tecniche di risoluzione di problemi SMT per formule senza quantificatori, ed utilizza euristiche per evitare l'istanziazione di istanze inutili ed eseguire test di soddisfabilibilità incrementali [23]. La tecnica è completa sotto alcune ipotesi su T_I e T_E [20]; se queste ipotesi non sono soddisfatte, la procedura di istanziazione può essere utilizzata correttamente senza perdita di precisione per verificare proprietà di safety, tuttavia vi è una maggior probabilità che l'algoritmo di raggiungibilità all'indietro non termini.

Le euristiche di istanziazione dei quantificatori sono complete in MCMT, mentre le corrispondenti euristiche di Yices non lo sono. Questo punto è abbastanza delicato: per fare un esempio, si prenda un protocollo di mutua esclusione (ideato da Szymanski) che può essere considerato un tipico esempio di problema riguardante sistemi parametrici. Per verificarne la safety MCMT genera 5196 problemi SMT, di cui 1153 soddisfacibili mentre 4043 insoddisfacibili. Mentre la procedura di instanziazione dei quantificatori implementata in MCMT associata a Yices permette di risolvere tutti i 5196 problemi SMT, Yices da solo (con la sua procedura interna per la gestione dei quantificatori) non riesce a risolvere i 1153 problemi soddisfacibili, per i quali ritorna "*unknown*", mentre per gli altri 4043 insoddisfacibili ritorna "*unknown*" per 1820 e riesce a risolvere i restanti 2223.

2.3.1 Euristiche per la sintesi degli invarianti

MCMT supporta la sintesi degli invarianti, ed implementa tecniche di astrazione e di accelerazione. La sintesi degli invarianti è stata introdotta in [21]. MCMT può generare invarianti con al più due variabili universalmente quan-

tificate: mentre viene eseguito l'algoritmo di raggiungibilità all'indietro per testare la safety del sistema, vengono generati alcuni candidati invarianti grazie ad alcune euristiche (si veda [21] per i dettagli) e successivamente viene eseguito l'algoritmo di raggiungibilità all'indietro (ponendo in questo caso un limite alle risorse disponibili) per verificare se i candidati invarianti devono essere scartati oppure possono essere tenuti. Gli invarianti ottenuti tramite questa procedura vengono utilizzati nella procedura principale di raggiungibilità all'indietro durante i test di punto fisso. La sintesi degli invarianti è effettuata da MCMT (anche) utilizzando una tecnica chiamata *astrazione della segnatura*: essa consiste nel proiettare (tramite un'eliminazione dei quantificatori, quando possibile) quei letterali contenenti un sottoinsieme delle variabili di tipo array. In questo modo si genera una sovra-approssimazione dell'insieme degli stati raggiungibili, tuttavia questo fatto non comporta una perdita di precisione, in quanto è utilizzato per la ricerca degli invarianti, mentre l'algoritmo di raggiungibilità all'indietro che testa la safety del sistema calcola gli stati raggiungibili all'indietro sempre in modo preciso.

Inoltre MCMT implementa tecniche di *astrazione* seguendo le direttive esposte in [8]: come già osservato, questa tecnica è utile per sistemi formalizzati con vincoli aritmetici, come ad esempio le reti di Petri.

Parallelizzazione della verifica degli invarianti

Per aumentare le prestazioni di MCMT è stato sviluppato uno script (in Python, [19]) che parallelizza la verifica degli invarianti: l'idea alla base di questa euristica è quella di sfruttare eventuali processori di macchine *multicore* per determinare se una formula φ (opportunamente trovata grazie ad alcune euristiche implementate in MCMT) rappresenta un invariante per un dato problema di safety $(\mathcal{S}, U)$ evitando che venga interrotta l'esecuzione dell'algoritmo di raggiungibilità a ritroso sul problema completo.

Sebbene dal lato teorico l'idea sembra buona, dal lato pratico si è visto che avere un alto numero di invarianti non aumenta la probabilità di terminazione dell'algoritmo di raggiungibilità a ritroso. Come verrà descritto nel Capitolo 7 in riferimento ad un protocollo analizzato, quello che conta non è il numero degli invarianti che vengono trovati, bensì il tipo di informazioni che vengono fornite ad MCMT tramite gli invarianti.

2.4 Il modello *stopping failures*

In un sistema stopping failure un processo può fallire scomparendo dal sistema in qualsiasi istante e non ritornando più in attività. È opportuno introdurre il modello stopping failure in un sistema a variabili array ad un livello puramente sintattico.

Sia $(\mathcal{S}, U)$ un problema di safety dove $\mathcal{S} = (a, I, \tau)$ è un sistema a variabili array e $U(a)$ è una formula che descrive l'insieme degli stati unsafe; il *problema di safety per modelli stopping failures* $(\tilde{\mathcal{S}}, \tilde{U})$ associato a $(\mathcal{S}, U)$ può essere costruito come segue.

(a) Innanzitutto bisogna espandere la segnatura per poter utilizzare una formula $\mathtt{C}(\mathtt{rash})(i, a[i])$ che descriva i processi falliti. Il modo più facile per raggiungere questo scopo è aggiungere una variabile booleana che indichi se un processo è fallito o meno. Tuttavia, dato che per semplicità i sistemi introdotti hanno una sola variabile, è necessaria una trasformazione un po' più complessa. Al posto della teoria T_E mono-sorta viene utilizzata una teoria $\tilde{T}_E$ a tre sorti i cui modelli sono il prodotto cartesiano dei modelli di T_E e dei due elementi dell'algebra booleana $\{0, 1\}$. La segnatura Σ_E viene estesa con le costanti $1, 0$ e con le due operazioni di proiezione: ciò significa che nella segnatura estesa vi sono operazioni unarie pr_1 e pr_2 che indicano rispettivamente i 'vecchi' dati (cioè quelli che non vengono modificati da questa espansione della segnatura) ed il contenuto della 'nuova variabile booleana' (in seguito verrà scritto rispettivamente $e.1$ ed $e.2$ invece di $pr_1(e)$ e $pr_2(e)$). Quindi $\mathtt{C}(i, a[i])$ è ora definito come $a[i].2 = 0$ e il predicato complementare $\mathtt{A}(i, a[i])$ è definito come $a[i].2 = 1$; la scrittura $\mathtt{A}(i, a[i])$ indica che 'il processo i è attivo (quando il sistema è nello stato a)'. Sia $\tilde{A}_I^E$ la teoria risultante dalla sostiuzione di T_E con $\tilde{T}_E$.

(b) Per costruire le formule $\tilde{I}(a)$ e $\tilde{U}(a)$ basta prendere $I(a)$ e $U(a)$, sostituire i termini t di sorta $\mathtt{ELEM}$ con $t.1$, e relativizzare i quantificatori degli indici alla formula $\mathtt{A}$. In particolare, una $\exists^I$-formula $\exists \underline{i}\, \phi(\underline{i}, a[\underline{i}])$ diventa $\exists \underline{i}\, (\mathtt{A}(\underline{i}, a[\underline{i}]) \wedge \phi(\underline{i}, a[\underline{i}].1))$ (qui, se $\underline{i} = i_1, \ldots, i_n$, l'abbreviazione $\mathtt{A}(\underline{i}, a[\underline{i}])$ sta per $\bigwedge_{k=1}^{n} \mathtt{A}(i_k, a[i_k])$). Similmente, una $\forall^I$-formula $\forall \underline{i}\, \phi(\underline{i}, a[\underline{i}])$ diventa $\forall \underline{i}\, (\mathtt{A}(\underline{i}, a[\underline{i}]) \rightarrow \phi(\underline{i}, a[\underline{i}].1))$.

(c) Per descrivere la trasformazione delle formule di transizione, bisogna innanzitutto trasformare una funzione definita per casi di sorta `ELEM` che presenta un argomento j di sorta `INDEX` e un argomento d di sorta `ELEM`.

Sia $F(\underline{i}, \underline{e}, j, d)$ tale funzione[5] e la sua definizione sia della forma

$$\begin{aligned} F(\underline{i}, \underline{e}, j, d) \ := \ \texttt{case of} \ \ \{ & \\ & C_1(\underline{i}, \underline{e}, j, d) \ \ : t_1(\underline{i}, \underline{e}, j, d) \\ & \qquad \cdots \\ & C_k(\underline{i}, \underline{e}, j, d) \ \ : t_k(\underline{i}, \underline{e}, j, d) \\ \}. & \end{aligned}$$

$\tilde{F}(\underline{i}, \underline{e}, j, d)$ è definita aggiungendo un ulteriore caso come segue:

$$\begin{aligned} \tilde{F}(\underline{i}, \underline{e}, j, d) \ := \ \texttt{case of} \ \{ & \\ & d.2 = 0 & : \langle d.1, 0 \rangle \\ & d.2 = 1 \wedge C_1(\underline{i}, \underline{e}.1, j, d.1) & : \langle t_1(\underline{i}, \underline{e}.1, j, d.1), 1 \rangle \\ & \qquad \cdots & \\ & d.2 = 1 \wedge C_k(\underline{i}, \underline{e}.1, j, d.1) & : \langle t_k(\underline{i}, \underline{e}.1, j, d.1), 1 \rangle \\ \}. & \end{aligned}$$

(d) Supponiamo ora che la formula di transizione $\tau(a, a')$ è costruita a partire da Σ-formule atomiche e equazioni funzionali $a' = \lambda j.F(\underline{i}, a[\underline{i}], j, a[j])$ (che coinvolgono ulteriori funzioni F definite per casi) alle quali vengono applicati operatori booleani e quantificatori di sorta `ELEM` e `INDEX`. Per costruire $\tilde{\tau}(a, a')$ innanzitutto vengono sostituite le funzioni definite per casi F da corrispondenti $\tilde{F}$, i termini t di sorta `ELEM` da $t.1$, e vengono relativizzati tutti i quantificatori relativi a simboli di sorta `INDEX` rispetto alla formula `A`. Infine vengono aggiunte alle transizioni modificate ulteriori disgiunzioni

$$\exists i \ (a' = \lambda j.(\texttt{if} \ i = j \ \texttt{then} \ \langle a[j].1, 0 \rangle \ \texttt{else} \ a[j])) \tag{2.5}$$

[5] Per semplicità e senza perdita di generalità, assumiamo che la funzione definita per casi viene introdotta ad un livello della segnatura originale Σ e non su ulteriori estensioni definite per casi di essa.

per indicare che i processi possono fallire in ogni momento.

Secondo le trasformazioni descritte, le disgiunzioni (2.4) delle transizioni in forma funzionale diventano

$$\exists \underline{i}\,(\mathtt{A}(\underline{i}, a[\underline{i}]) \wedge \phi_L(\underline{i}, a[\underline{i}].1) \wedge a' = \lambda j.\tilde{F}_G(\underline{i}, a[\underline{i}], j, a[j])) \tag{2.6}$$

Invece le disgiunzioni (2.4) delle transizioni con guardie universali espresse in forma funzionale diventano

$$\exists \underline{i}\,(\mathtt{A}(\underline{i}, a[\underline{i}]) \wedge \phi_L(\underline{i}, a[\underline{i}].1) \wedge \forall k\,(\mathtt{A}(k, a[k]) \rightarrow \psi(\underline{i}, k, a[\underline{i}].1, a[k].1)) \wedge a' = \\ = \lambda j.\tilde{F}_G(\underline{i}, a[\underline{i}], j, a[j])) \tag{2.7}$$

L'intuizione dietro alle (2.6) e (2.7) è che i processi falliti escono dal sistema e non svolgono più alcuna azione, e solo i processi non falliti possono permettere o impedire l'applicazione di una transizione.

2.5 Risultati di comparazione

Da ultimo è opportuno studiare la relazione tra un problema di safety $(\mathcal{S}, U)$ ed il corrispondente problema di safety $(\tilde{\mathcal{S}}, \tilde{U})$ supponendo il modello stopping failure.

Proposizione 2.5.1. *Se $\tilde{\mathcal{S}}$ è safe rispetto ad $\tilde{U}$, allora $\mathcal{S}$ è safe rispetto ad U.*

Dimostrazione. Per assurdo sia

$$I(a_0) \wedge \tau(a_0, a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge U(a_n)$$

A_I^E-soddisfacibile in un modello $\mathcal{M}$. Si consideri il corrispondente modello $\tilde{\mathcal{M}}$ per $\tilde{A}_I^E$ e si assegni alle variabili array a_k $(k = 0, \ldots, n)$ la funzione i cui valori in corrispondenza dell'indice i sono la coppia formata dal precedente valore $a(i)$ e dal valore booleano 1; sia $\tilde{a}_k$ l'array così definito (quindi vale $\tilde{\mathcal{M}} \models \bigwedge_k \forall i\, \mathtt{A}(i, \tilde{a}_k[i])$). Dalle definizioni (a)-(b)-(c)-(d) della Sezione 2.4, si può vedere facilmente che vale

$$\tilde{\mathcal{M}} \models I(\tilde{a}_0) \wedge \tau(\tilde{a}_0, \tilde{a}_1) \wedge \cdots \wedge \tau(\tilde{a}_{n-1}, \tilde{a}_n) \wedge U(\tilde{a}_n)$$

quindi $\tilde{\mathcal{S}}$ non è safe rispetto ad $\tilde{U}$. □

Ciò dimostra che una prova di safety ottenuta supponendo il modello stopping failures implica la safety del sistema originale. Ovviamente non è vero il contrario, a meno che le transizioni del sistema siano in una forma funzionale. Da qui in poi, per *formula inziale* I si intenderà una $\forall^I$-formula e per *formula unsafe* $U(a)$ si intenderà una $\exists^I$-formula.

Lemma 2.5.1. *Supponiamo che* $\tau(a, a')$ *sia in forma funzionale. Sia* K *una* $\exists^I$*-formula e sia* K_0 *la* $\exists^I$*-formula* $Pre(\tau, K)$.[6] *Allora le* $\exists^I$*-formule* $\tilde{K} \vee \tilde{K}_0$ *e* $\tilde{K} \vee Pre(\tilde{\tau}, \tilde{K})$ *sono* $\tilde{A}^I_E$*-equivalenti.*

Dimostrazione. Seguono dai punti (b)-(c)-(d) della Sezione 2.4 (in particolare, dal fatto che i processi falliti non possono più eseguire alcuna azione nel sistema). □

Proposizione 2.5.2. *Se la segnatura* Σ_I *è relazionale e* τ *è in forma funzionale, allora* $\tilde{\mathcal{S}}$ *è safe rispetto ad* $\tilde{U}$ *se e solo se* $\mathcal{S}$ *è safe rispetto ad* U*.*

Dimostrazione. La non safety di $\tilde{\mathcal{S}}$ rispetto ad $\tilde{U}$ significa che

$$\bigvee_{k \leq n} Pre^k(\tilde{\tau}, \tilde{U})$$

è $\tilde{A}^E_I$-consistente con $\tilde{I}$ per qualche n. Ora se si pone

$$K := \bigvee_{k \leq n} Pre^k(\tau, U)$$

si ottiene che $\tilde{K} \wedge \tilde{I}$ è $\tilde{A}^E_I$-consistente se e solo se $K \wedge I$ è A^E_I-consistente (si veda il Lemma seguente). Sfruttando il Lemma 2.5.1 si ottiene la tesi. □

Lemma 2.5.2. *Sia* Σ_I *una segnatura relazionale. Sia* I *una* $\forall^I$*-formula e sia* K *una* $\exists^I$*-formula. Allora* $\tilde{I} \wedge \tilde{K}$ *è* $\tilde{A}^E_I$*-consistente se e solo se* $I \wedge K$ *è* A^E_I*-consistente.*

[6] È bene ricordare che le $\exists^I$-formule sono chiuse sotto il calcolo della preimmagine nel caso in cui le transizioni sono in forma funzionale (si veda la Sezione 2.2.1). È anche chiaro che le $\exists^I$-formule sono chiuse sotto disgiunzione (il bisogno della formula disgiunta $\tilde{K}$ nell'enunciato del Lemma è dovuto al fatto per il quale $\tilde{\tau}$ contiene anche la disgiunzione (2.5)).

Dimostrazione. Da un lato la dimostrazione è banale. D'altro lato, si supponga che $\tilde{\mathcal{M}} \models \tilde{I} \wedge \tilde{K}$ e sia $K := \exists \underline{i} \phi(\underline{i}, a[\underline{i}])$. Allora $\tilde{\mathcal{M}} \models \tilde{I} \wedge \mathtt{A}(\underline{i}, a[\underline{i}]) \wedge \phi(\underline{i}, a[\underline{i}])$ per qualche assegnamento di a e di $\underline{i}$. Il A_I^E-modello $\mathcal{M}$ sia ottenuto dall' $\tilde{A}_I^E$-modello $\tilde{\mathcal{M}}$ restringendo l'interpretazione della sorta `INDEX` agli elementi assegnati ad $\underline{i}$. È chiato che tutti i processi selezionati per $\mathcal{M}$ sono attivi in $\tilde{\mathcal{M}}$ (cioè $\tilde{\mathcal{M}} \models \mathtt{A}(j, a[j])$ vale per ogni $j \in \mathtt{INDEX}^{\mathcal{M}}$), quindi se si assegna in $\mathcal{M}$ ad a l'array che è stato ristretto nel dominio, da $\tilde{\mathcal{M}} \models \tilde{I} \wedge \phi(\underline{i}, a[\underline{i}])$ si ottiene $\mathcal{M} \models I(a) \wedge K(a)$. □

A questo punto è possibile mostrare come eliminare le guardie universali dalla versione stopping failures di un sistema che possiede transizioni espresse in forma funzionale contenenti quardie universali. Sia $\mathcal{S} = (a, I, \tau)$ tale sistema e sia $\tilde{\mathcal{S}} = (a, \tilde{I}, \tilde{\tau})$ la versione stopping failure dello stesso (quindi i disgiunti di $\tilde{\tau}$ hanno la forma (2.7)). Si costruisca un terzo sistema $\mathcal{S}_+ = (a, I_+, \tau_+)$. La formula iniziale I_+ è $\tilde{I}$; per ottenere τ_+, si tiene (2.5) e gli altri disgiunti (2.7) di $\tilde{\tau}$ sono così modificati:

$$\exists \underline{i} \, (\mathtt{A}(\underline{i}, a[\underline{i}]) \wedge \phi_L(\underline{i}, a[\underline{i}].1) \wedge a' = \lambda j. F_G^+(\underline{i}, a[\underline{i}], j, a[j])) \tag{2.8}$$

dove le funzioni definite in casi F_G^+ sono ottenute come segue. Supponiamo che F_G abbia la forma

$$\begin{aligned} F_G(\underline{i}, \underline{e}, j, d) \; := \; \mathtt{case\ of} \;\; \{ & \\ & C_1(\underline{i}, \underline{e}, j, d) \;\; : t_1(\underline{i}, \underline{e}, j, d) \\ & \quad \cdots \\ & C_k(\underline{i}, \underline{e}, j, d) \;\; : t_k(\underline{i}, \underline{e}, j, d) \\ \}; & \end{aligned}$$

si definisce $F_G^+(\underline{i}, \underline{e}, j, d)$ aggiungendo un nuovo caso come segue

$$\begin{aligned} F_G^+(\underline{i}, \underline{e}, j, d) \; := \; & \mathtt{caseof} \; \{ \\ & d.2 = 0 \vee \neg \psi(\underline{i}, j, \underline{e}, d.1) && : \langle d.1, 0 \rangle \\ & d.2 = 1 \wedge \psi(\underline{i}, j, \underline{e}, d.1) \wedge C_1(\underline{i}, \underline{e}.1, j, d.1) && : \langle t_1(\underline{i}, \underline{e}.1, j, d.1), 1 \rangle \\ & \cdots \\ & d.2 = 1 \wedge \psi(\underline{i}, j, \underline{e}, d.1) \wedge C_k(\underline{i}, \underline{e}.1, j, d.1) && : \langle t_k(\underline{i}, \underline{e}.1, j, d.1), 1 \rangle \\ & \}. \end{aligned}$$

In altre parole, nella (2.8) le guardie universali sono state rimosse, ma i processi che la violano sono considerati falliti. Le motivazioni per le quali $\tilde{\mathcal{S}}$ e $\mathcal{S}_+$ sono equivalenti sono che la transizione τ_+ può essere simulata da un numero finito di fallimenti seguiti dall'applicazione della transizione $\tilde{\tau}$. Per rendere questa intuizione precisa, è necessaria la nozione di *bisimulazione*. Uno *stato* di A_E^I è un array s (ossia un elemento di $\texttt{ARRAY}^{\mathcal{M}}$) in un modello $\mathcal{M}$ of A_E^I con un numero finito di indici.[7] L'n-esima iterazione di $\tau(a, a')$ è cosí definita:

(i) $\tau^0(a, a')$ è $a = a'$;

(ii) $\tau^{n+1}(a, a')$ è $\exists a''(\tau(a, a'') \wedge \tau^n(a'', a'))$.

Definizione 2.5.1. Siano $\mathcal{S}_1 = (a, I_1, \tau_1)$ e $\mathcal{S}_2 = (a, I_2, \tau_2)$ due sistemi a variabili array con la stessa teoria di background A_E^I. Una *bisimulazione* tra $\mathcal{S}_1$ e $\mathcal{S}_2$ è una relazione R tra gli stati di A_E^I che soddisfa le seguenti proprietà (la notazione $\mathcal{M}_s$ indica il modello con indici finiti da cui è preso s):

- per ogni s_1 esiste un s_2 tale che vale $R(s_1, s_2)$ (e viceversa);
- se vale $R(s_1, s_2)$ allora $\mathcal{M}_{s_1} \models I_1(s_1)$ se e solo se $\mathcal{M}_{s_2} \models I_2(s_2)$;
- se vale $R(s_1, s_2)$ e $\mathcal{M}_{s_1} \models \tau(s_1, s_1')$ allora esistono un s_2' ed un $n \geq 0$ tali che valgono $R(s_1', s_2')$ e $\mathcal{M}_{s_2} \models \tau^n(s_2, s_2')$ (e viceversa).

I problemi di safety $(\mathcal{S}_1, U_1)$ e $(\mathcal{S}_2, U_2)$ sono compatibili con la bisimulazione R se e solo se $R(s_1, s_2)$ implica che ($\mathcal{M}_{s_1} \models U_1(s_1)$ se e solo se $\mathcal{M}_{s_2} \models U_2(s_2)$).

Il seguente lemma è immediatamente dedotto dal fatto che (2.1) è un $\exists^{A,I}\forall^I$-enunciato e che un $\exists^{A,I}\forall^I$-enunciato è A_I^E-soddisfacibile se e solo se è soddisfacibile in un modello ad indici finiti (si veda [20] per una dimostrazione):

Lemma 2.5.3. *Sia R una bisimulazione tra $\mathcal{S}_1 = (a, I_1, \tau_1)$ e $\mathcal{S}_2 = (a, I_2, \tau_2)$ tale che i problemi di safety $(\mathcal{S}_1, U_1)$ e $(\mathcal{S}_2, U_2)$ sono compatibili con R. Allora U_1 è safe per $\mathcal{S}_1$ se e solo se U_2 è safe per $\mathcal{S}_2$.*

[7] Un modello $\mathcal{M}$ con un numero finito di indici è un modello nel quale il supporto di $\texttt{INDEX}^{\mathcal{M}}$ è un insieme finito.

Utilizzando la relazione d'identità come bisimulazione, dal Lemma appena enunciato è facile dedurre che:

Teorema 2.5.1. *Sia $(\mathcal{S}, U)$ un problema di safety per il sistema a variabili array $\mathcal{S}$ che possiede transizioni con guardie universali espresse in forma funzionale. Allora $\tilde{U}$ è safe per $\tilde{\mathcal{S}}$ se e solo se $\tilde{U}$ è safe per il sistema $\mathcal{S}_+$.*

Si noti che la transizione di τ_+ di $\mathcal{S}_+$ è in forma funzionale (cioè non ha quantificatori universali nella guardia).

Remark. MCMT esegue automaticamente la trasformazione da τ a τ_+. Tuttavia, le cose sono un po' più complicate perché le $\exists^I$-formule che MCMT usa per descrivere gli stati raggiungibili a ritroso da altri stati sono mantenute in forma primitiva e differenziata). Senza perdita di generalità si può assumere che la disgiunzione delle transizioni con guardie universali espresse in forma funzionale abbia la forma

$$\exists \underline{i}\,(\phi_L(\underline{i}, a[\underline{i}]) \wedge \forall j\,(j \notin \underline{i} \rightarrow \psi(\underline{i}, j, a[\underline{i}], a[j])) \wedge a' = \lambda j. F_G(\underline{i}, a[\underline{i}], j, a[j])) \quad (2.9)$$

dove $j \in \underline{i}$ è la formula che indica che j è uguale ad una delle $\underline{i}$ (in effetti, vincoli riguardanti le $\underline{i}$ possono essere direttamente inseriti in ϕ_L). Il formato di input dei file MCMT assume implicitamente che il quantificatore $\forall j$ delle sottoformule universali della guardia è relativo ai processi non inclusi in $\underline{i}$. La formula ψ deve essere espressa in forma normale disgiuntiva con disgiunzioni mutualmente inconsistenti[8]. - questi disgiunti sono inseriti dall'utente uno per linea utilizzando la parola chiave `:uguard`. Durante la definizione di F_G^+, i casi potrebbero essere moltiplicati (e non solamente aumentati di 1) da MCMT, perché MCMT mantiene la specifica dei case come congiunzione di letterali (e non sono come formule senza quantificatori). In pratica, MCMT esegue la trasformazione in modo tale che il primo caso corrisponda ai casi $j \in \underline{i}$, gli ultimi casi sono casi di fallimento e i casi intermedi sono il risultato della combinazione di un disgiunto di ψ e un caso di F_G. Casi inconsistenti che potrebbero sorgere in questo modo non vengono eliminati, ma non verranno mai applicati dato che le $\exists^I$-formule risultanti dal calcolo della preimmagine devono soddisfare un test di consistenza eseguito attraverso l'SMT-solver.

[8] MCMT opera correttamente anche nel caso le disgiunzioni non sono mutualmente esclusive, ma potrebbe eseguire del lavoro ridondante se questa limitazione non viene rispettata

Remark. Tutti i quantificatori degli indici nelle formule $\tilde{I}$, $\tilde{U}$ sono relativizzati; seguendo una strada simile a quella tracciata dal 2.5.1, è facile mostrare che l'algoritmo per la ricerca a ritroso della Figura 2.1, una volta applicato a $\mathcal{S}_+$ e $\tilde{U}$, può solo produrre $\exists^I$-formule con quantificatori sugli indici relativizzati. Quindi, MCMT per semplificare la notazione e per aumentare la leggibilità assume che *tutte le quantificazioni sugli indici siano relativizzate*; di conseguenza, le formule $\mathtt{A}(i, a[i])$ non vengono mai mostrate esplicitamente (in particolare, non appaiono nel file prodotto da MCMT utilizzando l'opzione `-r`).

3 Un traduttore per MCMT

Il linguaggio accettato da MCMT è relativamente a basso livello e formalizzare direttamente in esso protocolli complessi con decine di transizioni che coinvolgono molte variabili risulta difficile e la probabilità di sbagliare è alta. La definizione di un linguaggio ad alto livello e lo sviluppo del relativo traduttore per generare automaticamente file di specifica accettati da MCMT permette un risparmio di tempo in fase di codifica delle specifiche, la formalizzazione diviene più leggibile e, in terzo luogo, la maggior parte degli errori sintattici e semantici viene intercettata dal traduttore e notificata all'utente.

Il capitolo è così organizzato: dopo aver introdotto brevemente l'architettura generale di un compilatore verranno descritte le diverse fasi di analisi del traduttore realizzato; innanzitutto verrà fatto un elenco dei *token* riconosciuti dall'analizzatore lessicale, successivamente verrà riportata la grammatica alla base dell'analisi sintattica e si farà un accenno all'analisi semantica. Esempi di formalizzazioni scritte secondo la sintassi di questo nuovo linguaggio possono essere rintracciati in Appendice A.

3.1 Architettura di un compilatore

Un compilatore si divide in due parti principali, una prima parte detta *front-end* svolge l'*analisi* del codice sorgente, ed una seconda parte detta *back-end* implementa gli algoritmi di *sintesi* del codice oggetto [6]. L'analisi viene

eseguita da diverse componenti logiche, che collaborano al fine di produrre una rappresentazione intermedia del codice sorgente; queste componenti sono gli analizzatori lessicale, sintattico e semantico.

Una volta ottenuta una rappresentazione intermedia del codice sorgente è possibile eseguire diverse ottimizzazioni e infine generare il codice oggetto.

Nel caso del traduttore descritto in questo capitolo, il *back-end* è privo di ottimizzazioni, e presenta solo la parte di generazione del codice oggetto.

3.1.1 Analisi del codice sorgente

La fase di analisi di un traduttore ha come scopo l'organizzazione del codice sorgente in una qualche struttura dati che permetta di controllarne efficientemente la sintassi, verificare l'assenza di errori semantici e facilitarne la traduzione. Tipicamente il sorgente viene organizzato in una struttura ad albero, detta *Albero di Sintassi Astratto* (*AST - Abstract Syntax Tree*).

Più precisamente, il processo di analisi si suddivide in tre fasi:

1. *Analisi lessicale* - Attraverso un analizzatore lessicale, chiamato anche *scanner*, il traduttore divide il codice sorgente in tanti "pezzetti" chiamati *token*. I *token* sono gli elementi minimi (non ulteriormente divisibili) di un linguaggio, ad esempio parole chiave (`existential`, `transition`), nomi di variabili , operatori ($+$, $-$, ecc...).

2. *Analisi sintattica* - L'analisi sintattica prende in ingresso la sequenza di token generata nella fase precedente ed esegue il controllo sintattico utilizzando le regole di produzione di una grammatica. Il risultato di questa fase è un albero di sintassi astratto.

3. *Analisi semantica* - L'analisi semantica si occupa di controllare il significato delle istruzioni presenti nel codice in ingresso. Test tipici di questa fase sono il *type checking*, la verifica che gli identificatori siano stati dichiarati prima di essere utilizzati, ecc. Come supporto a questa fase viene creata (durante l'analisi lessicale o, al più, quella sintattica) una tabella dei simboli (*symbol table*) che contiene informazioni su tutti gli elementi simbolici incontrati durante la costruzione dell'*AST*.

3.1.2 Sintesi del codice oggetto

Una volta prodotta una rappresentazione interna ad albero del codice sorgente, viene effettuata la sintesi del codice oggetto. Il back-end di un compilatore, oltre ad effettuare la traduzione del sorgente, implementa anche tecniche di ottimizzazione del codice: quando l'oggetto della compilazione è codice eseguibile, le tecniche di ottimizzazione possono essere sia indipendenti che dipendenti dall'architettura sulla quale verrà eseguito il codice prodotto. Dato che il traduttore realizzato non genera codice eseguibile, verrà tralasciata la descrizione delle tecniche di ottimizzazione e si rimanda ad altre fonti (ad esempio [6]) per ulteriori dettagli su questo argomento.

3.2 Descrizione del traduttore realizzato

3.2.1 Analisi lessicale

Un *analizzatore lessicale* è un programma che prende in input uno stream di caratteri (il codice sorgente da tradurre) e genera una sequenza di oggetti detti *token*. Ogni *token* può essere pensato come una coppia $(tokenId; attributo)$. I $tokenId$ sono utilizzati dall'analizzatore sintattico per costruire l'AST, mentre gli attributi sono i "valori reali" assunti dal token. Ad esempio, se nel codice viene incontrato il numero `8`, lo scanner passerà all'analizzatore sintattico (anche detto *parser*) la coppia `(NUMERO, 8)`. Nella costruzione dell'albero sintattico non importa tanto sapere che in quel punto del codice vi è il valore `8`, basta sapere che lo scanner ha trovato un `NUMERO`. Tuttavia l'analisi semantica potrebbe basarsi sul fatto che vi sia proprio `8` per controllare, ad esempio, che non vengano eseguite divisioni per 0.

Ogni *tokenId* di interesse per il linguaggio per il quale si vuole scrivere un compilatore può essere associato ad un'espressione regolare. Un'espressione regolare individua un linguaggio regolare (tipo 3 secondo la gerarchia di Chomsky). I linguaggi regolari sono riconosciuti da automi a stati finiti e si può dimostrare che da ogni espressione regolare è possibile generare un automa a stati finiti equivalente [27]. I token che si vogliono riconoscere sono necessariamente stringhe appartenenti a qualche linguaggio regolare. Ad esem-

pio i numeri sono parole del linguaggio regolare $L = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$, le stringhe di caratteri sono anch'esse parole di un linguaggio regolare.

Una volta individuate le espressioni regolari che definiscono la "forma" dei token da riconoscere, per generare uno scanner bisogna definire un diagramma di transizione, ottenibile combinando tutti gli automi a stati finiti derivati dalle espressioni regolari dei token. Ogni arco del diagramma di transizione è etichettato con un simbolo (o un insieme di simboli); quando viene letto un carattere del codice sorgente, il diagramma di transizione segue l'arco etichettato con quel carattere e passa in un nuovo stato. La scansione dell'input continua finché non vi sono più transizioni corrispondenti al simbolo letto; successivamente se lo stato nel quale si trova il diagramma di transizione è accettante, viene segnalato al parser che è stato riconosciuto un token e il diagramma torna allo stato iniziale. Viceversa viene sollevata un'eccezione all'utente.

Nella seguente tabella sono riportati i token identificati per il linguaggio sviluppato e le rispettive espressioni regolari per riconoscerli.

Nome	Attributo	Espressione regolare riconosciuta
PIU		"+"
MENO		"−"
PER		"*"
DIVISO		"/"
ASSIGN		":="
TONDA_APERTA		")"
TONDA_CHIUSA		"("
QUADRA_APERTA		"]"
QUADRA_CHIUSA		"["
GRAFFA_APERTA		"{"
GRAFFA_CHIUSA		"}"
AND		"AND"
OR		"OR"
NOT		"!"
IMPLICA		"=>"

MIN		“<”
MAGG		“>”
MINUG		“<=”
MAGGUG		“>=”
UG		“=”
DIV		“! =”
VIRGOLA		“,”
PUNTOVIRGOLA		“;”
DUEPUNTI		“:”
TRUE		“true”
FALSE		“false”
LOCAL		“local”
GLOBAL		“global”
INDEX		“index”
TYPE		“define − type”
FUNC		“define − func”
PRED		“define − pred”
INIT		“initial”
UNSAFE		“unsafe”
TRANSITION		“transition”
GUARD		“guard”
UGUARD		“uguard”
UPDATE		“update”
LAMBDA		“lambda”
EXISTENTIAL		“existential”
UNIVERSAL		“universal”
AXIOM		“axiom”
SYSTEM_AXIOM		“system_axiom”
EEVAR		“eevar”
VAR		“var”
INT		“int”
REAL		“real”

NAT		"nat"
BOOL		"bool"
CASE		"case"
OTHERWISE		"otherwise"
NUMERO	Numero naturale	$[0-9]^+$
REALE	Numero reale (non negativo)	$[0-9]^+$"."$[0-9]^+$
IDENT	Sequenza di numeri e lettere che comincia con una lettera	$[a-zA-Z][a-zA-Z0-9]^*$
MCMT_STRINGA	Stringa di caratteri utilizzata per indicare opzioni di MCMT	Riconosciuto tramite lo stato ⟨SMT⟩

flex: The Fast Lexical Analyzer

Il tool *flex* [33] è un *generatore di analizzatori lessicali* che produce il codice di uno scanner partendo dalla definizione dei token ed espressioni regolari ad essi associate. *flex* prende in input una lista di espressioni regolari ognuna rappresentante un *token* da riconoscere ed associata ad un'azione da eseguire; esso genera a partire da questa lista un diagramma di transizione e dà in output il file `lex.yy.c` contenente il codice `c` della funzione `yylex()` che ne simula il comportamento. Il file `lex.yy.c` può essere compilato[1] per produrre un eseguibile, il cui compito è analizzare il flusso di caratteri e riconoscere le sequenze rappresentate dalle espressioni regolari. Per ogni corrispondenza trovata verrà eseguito il codice `c` associato a quella determinata espressione regolare. Il codice da eseguire quando viene riconosciuto un token può riguardare la compilazione della tabella dei simboli, l'allocazione della memoria per particolari oggetti, ecc...

[1] Il compilatore utilizzato è `gcc` [18].

3.2.2 Analisi sintattica

La seconda componente logica di un traduttore è rappresentata dall'analizzatore sintattico, o *parser*. Il compito del parser è quello di dare una struttura sintattica ai token riconosciuti dallo scanner.

Per realizzare un analizzatore sintattico è fondamentale definire la grammatica che genera le parole del linguaggio da riconoscere. Le grammatiche introdotte da Chomsky sono lo strumento adatto per costruire i compilatori perché principalmente forniscono una precisa e semplice specifica della sintassi del linguaggio di programmazione che si vuole utilizzare e consentono uno sviluppo iterativo del linguaggio di programmazione, permettendo di aggiungere di volta in volta nuovi costrutti per potenziare il linguaggio. Inoltre la struttura indotta dalla grammatica al codice sorgente facilita la traduzione e la ricerca degli errori, che sono i due compiti fondamentali di un traduttore. La grammatica inventata per il linguaggio in oggetto è la seguente:

specifica	→	seqIstr
seqIstr	→	istruzione
	|	seqIstr istruzione
istruzione	→	transizione
	|	dichiarazione
	|	iniziale
	|	unsafe
	|	assioma
	|	mcmt
assioma	→	**axiom** (seqIdentQuant) **{** formula **}**
dichiarazione	→	**index** dominio end
	|	**define-type** identificatore smtStringa
	|	**local** dominio identificatore [seqDom]
	|	**global** dominio identificatore [seqDom]
	|	**define-func** dominio identificatore (seqDom)
	|	**define-pred** identificatore (seqDom)
	|	**eevar** identificatore **:** dominio **;**
	|	**var** identificatore **:** dominio **;**
seqDom	→	dominio
	|	seqDom **,** dominio
iniziale	→	**initial** (seqIdentQuant) **{** formula **}**
unsafe	→	**unsafe** (seqIdentQuant) **{** formula **}**

invariant	→	**suggested_negated_invariant** (seqIdentQuant) **{** formula **}**
identAlone	→	identificatore
identDom	→	identificatore **:** dominio
seqIdentQuant	→	identQuant
	|	seqIdentQuant **,** identQuant
identQuant	→	**existential** identificatore **:** dominio
	|	**universal** identificatore **:** dominio
transizione	→	**transition** (seqIdentQuant) **{ guard:** formula nextTransizione **}**
nextTransizione	→	**update:** seqAssignment
	|	seqUguard formula **update:** seqAssignment
seqUguard	→	**uguard:** formula
	|	seqUguard **uguard:** formula
formula	→	letterale
	|	**!** formula
	|	formula **AND** formula
	|	formula **OR** formula
	|	formula => formula
	|	**(** formula **)**
seqAssign	→	assign
	|	seqAssign assign
termine	→	identificatore [seqTerm]
	|	identificatore (seqTerm)
	|	termine + termine
	|	+ termine
	|	termine **-** termine
	|	**-** termine
	|	termine ***** termine
	|	termine / termine
	|	**(** termine **)**
seqTerm	→	termine
	|	seqTerm **,** termine
individuale	→	reale
	|	numero
	|	identificatore
	|	boolean
letterale	→	termine > termine
	|	termine >= termine
	|	termine < termine
	|	termine <= termine

	|	termine = termine
	|	termine **!=** termine
assignment	→	assign
	|	assignLShort
	|	assignGShort
	|	lambdaAssign
seqAssignment	→	assignment
	|	seqAssignment assignment
assign	→	identificatore [identAlone] **:=** termine **;**
assignLShort	→	identificatore **:= lambda (** identDom **) {** termine **}**
assignGShort	→	identificatore **:=** termine **;**
lambdaAssign	→	**lambda (** identDom **;** letterale **) {** seqCase **otherwise:** seqAssign **}**
seqCase	→	**case** formula **:** seqAssign
	|	seqCase **case** formula **:** seqAssign
dominio	→	**int**
	|	**bool**
	|	**nat**
	|	**real**
	|	identificatore
boolean	→	**true**
	|	**false**
mcmt	→	**(:mcmt** stringa **:)**
identificatore	→	sequenza non vuota di lettere e cifre che inizia con una lettera
numero	→	sequenza non vuota di cifre
reale	→	numero **.** numero
stringa	→	sequenza di caratteri

Generazione dell'analizzatore sintattico - Bison

Per alcune tipologie di grammatiche esistono tool che permettono la generazione automatica del parser: il più famoso è senza dubbio *yacc* - *Yet Another Compiler-Compiler* [28], ma per il traduttore realizzato è stato utilizzato *Bi-*

son [32], una versione GNU di *yacc*. *Bison* è un *LALR parser generator*[2], ovvero un tool che prende in input una grammatica G (espressa nella forma di *Backus-Naur*, *BNF* [27]) scritta secondo una certa sintassi e produce il file `y.tab.c` contenente il codice `c` di un programma che implementa un parser LALR in grado di eseguire il parsing del codice sorgente. *Bison* può essere utilizzato assieme a *flex*, in questo modo il parser generato sfrutterà la funzione `yylex()` messa a disposizione dello scanner per prelevare i token dal codice sorgente; il parser stesso, a sua volta, rende disponibile la primitiva `yyparse()` utilizzabile per eseguire l'analisi del sorgente.

La generazione automatica del parser può rivelare se vi sono delle ambiguità nella grammatica, o altri problemi introdotti in fase di progettazione del linguaggio.

3.3 Sintesi del codice oggetto

Nel traduttore proposto la fase di sintesi è ridotta all'osso: dal momento che non vi sono ottimizzazioni, il codice oggetto viene generato direttamente dall'albero sintattico costruito dal parser.

3.4 Organizzazione del codice

Il traduttore è scritto in `c`, ed il codice del traduttore è così organizzato:

- `main.c` - il *main* del programma.
- `grammar.y` - file contenente le specifice della grammatica, compilato con *Bison*[3]

[2] La sigla LALR sta per LALR(1), *LookAhead Left-to-right Rightmost derivation*, ed individua i parser che leggono l'input da sinistra verso destra, che "guardano avanti" un carattere (senza consumarlo) per sapere quale regola di riduzione applicare, e che producono una derivazione destra del codice in input.

[3] La versione di *Bison* utilizzata è la 2.4.1, e la compilazione è stata effettuata con le opzioni `-dyv`.

- `lexical.l` - file contenente le specifiche dello scanner, compilato con *flex*[4]
- `common.h` - definizioni delle costanti che identificano il numero massimo di variabili, transizioni, formule, ecc...
- `data_structures/` - in questa directory vi sono i file per gestire le strutture dati relative ad assiomi, i vari *case* degli aggiornamenti delle transizioni, formule, funzioni, transizioni, symbol table, ecc...

[4]La versione di *flex* utilizza è la 2.5.35

4

Sistemi fault-tolerant

Con il termine "*fault-tolerant*" si identifica un sistema in grado di continuare ad operare correttamente nonostante si verifichino fallimenti ad una o piu' componenti del sistema stesso. I protocolli fault-tolerant sono ideati appositamente per gestire i fallimenti parziali in un sistema distribuito e permetterne il corretto funzionamento. Una particolarita' di questa classe di protocolli consiste nel fatto che sono progettati in modo da garantire il corretto funzionamento del sistema indipendentemente dal numero di processi coinvolti. Questi sistemi distribuiti vengono anche detti parametrici.

In questo capitolo verranno descritti brevemente i *sistemi tolleranti ai guasti* (sistemi *fault-tolerant*). Innanzitutto verrà definito cosa si intende per *guasto* (*fault*), *fallimento* (*failure*) ed *errore* (*errore*); in secondo luogo, dopo aver elencato i *modelli di fallimento* proposti in letteratura, ci si soffermerà sul problema del *Reliable Broadcast*.

4.1 Fallimenti, guasti ed errori

Se un sistema distribuito è progettato per fornire ai suoi utenti un certo numero di servizi, il sistema *fallisce* quando uno o più di questi servizi non possono essere (completamente) erogati [31]. Un *errore* (*error*) è una parte dello stato di un sistema che può portare ad un *fallimento* (*failure*). La causa dell'errore è chiamata *guasto* (*fault*). Guasti, errori e fallimenti sono quindi

legati dalla seguente relazione di casualità:

$$guasto \quad \rightarrow \quad errore \quad \rightarrow \quad fallimento$$

4.1.1 Modelli di fallimento

In letteratura (si veda, ad esempio, [31]) i fallimenti di una componente di un sistema informatico vengono così suddivisi:

- *crash* - la componente ha un comportamento conforme alle specifiche fino a quando smette di funzionare del tutto.
- *omission* - la componente fallisce occasionalmente nel rispondere ad un input esterno. Questi fallimenti si dividono in
  - *receive omission* - il fallimento riguarda la ricezione di input esterni;
  - *send omission* - la componente riceve ed elabora l'input, ma non invia correttamente l'output.
- *timing* - la risposta della componente è corretta, ma il tempo di risposta è al di fuori dell'intervallo specificato dalle specifiche.
- *response* - la risposta della componente ad un input non è corretta. L'errore può essere un *errore di valore*, oppure si può verificare uno *state transition failure*, ovvero si registra una deviazione dal corretto flusso d'esecuzione.
- *arbitrary* - la componente produce risposte arbitrarie ed è soggetta a fallimenti arbitrari nella temporizzazione.

Questi modelli di fallimento possono essere classificati in termini di "severità": il modello A è *più severo* del modello B se l'insieme dei comportamenti scorretti indicati da A è un sottoinsieme proprio dei comportamenti scorretti rappresentati da B [26]. Un protocollo che tollera i fallimenti di un modello B tollererà quindi i fallimenti di tutti i modelli meno severi di B. I fallimenti *crash* sono quelli più innocui e la ricerca del guasto in caso di fallimenti di

questi tipo è relativamente semplice; al contrario, i fallimenti *arbitrari* (detti anche *bizantini*) sono i più gravi e i guasti che li causano sono i più difficili da rintracciare e correggere.

4.2 Reliable Broadcast

Nei sistemi distribuiti costituiti da più processi che collaborano scambiandosi messaggi in rete, la tolleranza ai guasti si basa solitamente [26] sull'implementazione di un protocollo di *Reliable Broadcast* che offre delle primitive di rete per l'invio e ricezione dei messaggi al livello delle applicazioni (Figura 4.1). L'invio di un messaggio da parte dell'applicazione si traduce solitamente nell'esecuzione di diverse istruzioni da parte della componente che implementa il livello di *Reliable Broadcast*, e queste istruzioni potrebbero includere anche l'invio e ricezione di un certo numero di messaggi alle componenti *Reliable Broadcast* presenti sugli altri host del sistema. Per evitare un eccessivo carico computazionale e/o consumo di banda, il progettista del sistema individua innanzitutto il modello di fallimento "astratto" che più si avvicina a quello reale, e successivamente implementa un protocollo di *Reliable Broadcast* adatto per quel modello.

4.2.1 Definizione del problema

Definizione 4.2.1. Il problema del Reliable Broadcast è risolto quando il sistema soddisfa tre proprietà [13]:

- *validity*: se un processo p *corretto* invia in modalità broadcast un messaggio m, tutti i processi devono decidere, prima o poi, su di esso.
- *agreement*: se un processo p *corretto* decide su m, allora tutti i processi corretti devono decidere su m.
- *eventual decision*: Se un processo p invia in modalità broadcast un messaggio m, allora tutti i processi corretti devono prima o poi decidere su di esso oppure nessun processo corretto deve decidere.

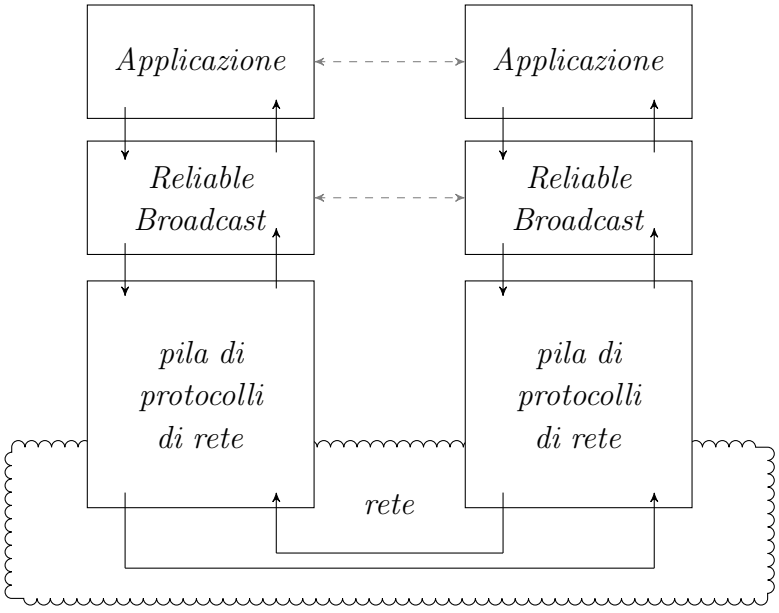

Figura 4.1: Organizzazione dello stack di rete con l'aggiunta di un livello che implementa un algoritmo di *Reliable Broadcast*.

La Definizione 4.2.1 non dice nulla sul comportamento dei processi *errati*; tuttavia è desiderabile, in certe situazioni, che anche i processi *errati* che raggiungono una decisione non siano in uno stato di inconsistenza rispetto a quelli corretti, cioè decidano sullo stesso valore. Se si pone anche questo vincolo sulla proprietà di *agreement* della Definizione 4.2.1, si giunge al problema dell'*Uniform Reliable Broadcast*:

Definizione 4.2.2. Il problema dell'*Uniform Reliable Broadcast* è risolto quando il sistema soddisfa le proprietà di *validity* e *eventual decision* descritte in Definizione 4.2.1, oltre alla seguente proprietà (di *uniform agreement*): se un processo p decide su m, allora tutti i processi corretti devono decidere su m.

4.3 Caratteristiche del sistema considerato

Per studiare il problema del *Reliable Broadcast*, il sistema distribuito in esame può essere rappresentato da n processi identificati univocamente da un *id* nell'intervallo $[1, n]$, che possono comunicare attraverso collegamenti affidabili in una rete *point-to-point* completamente connessa. Ogni processo p mantiene una variabile locale $\texttt{estimate}_p$ che rappresenta la stima corrente sulla quale verrà presa la decisione finale. I processi cambiano il valore della variabile `estimate` quando ricevono dei messaggi di *estimate* da parte del coordinatore. Il contenuto di questi messaggi rappresenta la nuova stima. Ciascun processo può trovarsi in due stati di decisione: *decided* o *undecided*. All'inizio dell'esecuzione dell'algoritmo ogni processo è nello stato *undecided*, e passa nello stato *decided* dopo aver ricevuto il primo messaggio di *decide*. Quando un processo riceve un messaggio di *decide* salva il valore della variabile $\texttt{estimate}_p$ in $\texttt{decisionValue}_p$. Una volta impostato il valore di $\texttt{decisionValue}_p$, il processo non lo cambia più (mentre può continuare a modificare il valore di $\texttt{estimate}_p$). Il sistema è sincronizzato: tutti i tempi necessari ai processi per eseguire le varie specifiche dell'algoritmo (ad esempio invio, ricezione, processing dei messaggi, ecc...) sono noti a priori.

Verranno considerate due tipologie di fallimento:

- *Crash failures*

- *Send-Omission failures*

Gli algoritmi analizzati (tratti da [13]) sono tutti basati sul paradigma dei "coordinatori rotanti": a turno, ogni processo ancora in esecuzione nel sistema diventa coordinatore. Ogni processo può essere coordinatore una ed una sola volta; quando tutti i processi non ancora falliti sono diventati coordinatori l'algoritmo termina. Il coordinatore cerca di far pervenire a tutti i processi indecisi la propria stima per permettere successivamente una decisione comune su di essa.

4.4 Formalizzazione e Verifica

Scopo del lavoro è eseguire una verifica parametrica (cioè indipendente dal numero di processi del sistema) dei protocolli proposti in [13]. Il traduttore descritto in Capitolo 3 verrà utilizzato per tradurre le formalizzazioni dei protocolli nella sintassi accettata da MCMT [22], tool attraverso il quale verrà eseguita la verifica.

4.4.1 Verifica

In questa tesi verrà presa in considerazione solo la proprietà di *safety* del problema del Reliable Broadcast, ovvero la proprietà secondo la quale *non vi saranno mai due processi corretti e decisi con due stime di decisione diverse*. Formalmente, verificare la *safety* di un problema di Reliable Broadcast significa verificare che nel sistema sia soddisfatta la proprietà di *agreement* della Definizione 4.2.1 oppure la proprietà di *uniform agreement* definita nella Definizione 4.2.2.

Le caratteristiche della macchina utilizzata per le verifiche sono

- Processore: Intel(R) Core(TM)2 Duo CPU E7300 @ 2.66GHz
- Memoria: 2 GB
- Sistema operativo: Debian

Tutti i tempi di verifica riportati nei prossimi capitoli sono espressi in secondi.

Parte II

Questa seconda parte della tesi mostra come MCMT può essere utilizzato per verificare protocolli che risolvono il problema del *Reliable Broadcast* assumendo diversi modelli di fallimento del sistema. Nella fattispecie, i protocolli che verranno analizzati non sono mai stati verificati formalmente.

Nel Capitolo 5 verrà preso in considerazione il modello di fallimento più semplice in ordine di severità [26], ovvero quello che ammette solo fallimenti *crash*. Per questo modello di fallimento verrà analizzato un protocollo proposto in [13]: dopo averlo formalizzato nel linguaggio descritto nel Capitolo 3, verrà eseguita una verifica parametrica nel numero dei processi coinvolti nel sistema utilizzando il tool MCMT. Un processo che fallisce *crash* di fatto non invia più alcun messaggio dopo un certo istante di tempo. Nel Capitolo 6 si cercherà di verificare lo stesso protocollo del capitolo precedente considerando un modello di fallimento più severo, per il quale i processi possono fallire omettendo di inviare solo alcuni messaggi: si mostrerà innanzitutto che quel protocollo non riesce a risolvere il problema del *Reliable Broadcast* assumendo il più ampio insieme di fallimenti appena descritto. Successivamente il protocollo in esame verrà modificato (come indicato in [13]) in modo tale che possa essere soddisfatta la proprietà di *agreement* anche con questa nuova tipologia di fallimento. Per ogni modifica effettuata, se MCMT indicherà che il protocollo è *unsafe*, verrà analizzata la sequenza di transizioni che rappresenta l'esecuzione che porta il sistema alla configurazione non voluta. Nel Capitolo 7 viene analizzato un protocollo (ottenuto modificando opportunamente quello descritto nel Capitolo 5) che riesce a risolvere il problema del *Reliable Broadcast* in caso di fallimenti *send-omission*.

5 Crash Failures

Questo capitolo è dedicato all'analisi ed alla verifica del protocollo riportato in [13] in grado di risolvere il problema del *Reliable Broadcast* assumendo solo fallimenti *crash*.

La prima parte del capitolo tratta l'analisi e la formalizzazione del protocollo mentre la seconda la verifica dello stesso utilizzando MCMT.

5.1 Modello di fallimento

Il primo protocollo analizzato risolve il problema del Reliable Broadcast in caso di fallimenti *crash*: secondo questa tipologia di fallimento si dà la seguente definizione:

Definizione 5.1.1. Un processo è *errato* se si ferma prematuramente rispetto alla normale conclusione dell'algoritmo. Prima di fermarsi, tuttavia, il processo ha eseguito correttamente ogni passo dell'algoritmo.

Ogni processo errato può fallire in qualsiasi istante, indipendentemente dai fallimenti degli altri processi del sistema. Nel modello di fallimento *crash* si assume anche la seguente nozione:

Definizione 5.1.2. Un processo è *corretto* se esegue tutto l'algoritmo senza mai fermarsi.

5.2 Formalizzazione

Algoritmo 5.2.1: Algoritmo per il Reliable Broadcast in caso di fallimenti *crash* ([13]).

1 **Inizializzazione**
2 $\texttt{estimate}_p \leftarrow \begin{cases} \texttt{m} & \text{se } p \text{ è il } \textit{sender} \ (m\text{: valore inizializzato dal sender}), \\ \perp & \text{altrimenti.} \end{cases}$
3 $\texttt{state}_p \leftarrow \texttt{undecided}$;
4
5 **for** $c \leftarrow 1, 2, \ldots, t+1$ **do**
`/* Il processo` c `diventa coordinatore per tre round */`
6 **Round 1**
7 Tutti i processi p *indecisi* inviano un messaggio di *request* a c
8 Se c non riceve alcun messaggio di *request* salta i round 2 e 3
9
10 **Round 2**
11 c invia in modalità broadcast $\texttt{estimate}_c$
12 Tutti i processi p *indecisi* che hanno ricevuto $\texttt{estimate}_c$ impostano $\texttt{estimate}_p \leftarrow \texttt{estimate}_c$
13
14 **Round 3**
15 c invia in modalità broadcast il messaggio *decide*
16 Tutti i processi p *indecisi* che hanno ricevuto il messaggio *decide* eseguono:
17 $\texttt{decision}_p \leftarrow \texttt{estimate}_p$
18 $\texttt{state}_p \leftarrow \texttt{decided}_p$
19
20

Il protocollo da formalizzare è riportato in Algoritmo 5.2.1; per definirlo formalmente sono necessarie le seguenti variabili, direttamente "importate" dallo pseudocodice dell'algoritmo:

- `estimate` - variabile locale booleana, indica la stima di ogni processo.
- `state` - variabile locale booleana, indica se un processo ha deciso.

- `decisionValue` - variabile locale booleana che indica il valore su cui ha deciso un processo (rappresenta la variabile `decision` dello pseudo-codice).

Inoltre bisogna indicare quale sia il coordinatore del sistema e quali processi hanno già ricoperto questa carica, per non eleggerli più volte. Per fare ciò servono altre due variabili:

- `coord` - variabile locale booleana: indica se un processo è coordinatore.
- `aCoord` - variabile locale booleana: indica se un processo è già stato coordinatore.

Serve anche una variabile che indichi in quale round si trova il sistema, ed un'altra variabile che segnali quali processi hanno già eseguito le operazioni del round:

- `round` - variabile globale a valore sui naturali che indica il round corrente.
- `done` - variabile locale booleana: indica se un processo ha terminato le operazioni del round.

I processi del sistema sono sincronizzati fra loro e quindi la formalizzazione deve contemplare questa sincronia: la variabile `done` è stata introdotta proprio per evitare che qualche processo passi al round successivo prima che qualcun'altro non abbia ancora finito di eseguire le operazioni del round. Il passaggio di round avverrà solo quando la variabile `done` sarà impostata per tutti i processi a `true`. Solo in questo caso tutti avranno eseguito le operazioni del round e quindi sarà lecito passare al successivo.

Infine, verrà utilizzata anche la variabile globale booleana `request`, che modella la ricezione da parte del coordinatore di un messaggio di *request*. Questa variabile è globale perché ai fini del protocollo non interessa conoscere i mittenti di tale messaggio, ma è sufficiente sapere se il coordinatore ha ricevuto o meno almeno un messaggio di questo tipo.

Le primitive di invio e ricezione dei messaggi in rete sono puramente implementative, e possono essere astratte con transizioni che operano sulle variabili locali `estimate` e `state` di ogni processo.

Prima di passare all'analisi delle transizioni è necessaria un'ultima nota: il modello di fallimento *stopping failures* assunto da MCMT permette di evitare l'introduzione di un'ulteriore variabile `faulty` che indichi se un processo è corretto o errato. È importante notare che questa "semplificazione" è possibile solo in questo caso, in quanto MCMT stesso gestisce i fallimenti *crash* dei processi. Qualora il modello di fallimento cambiasse, bisognerà modellare tramite opportune variabili il fallimento dei processi!

5.2.1 Processi

Nella formalizzazione, il valore locale di una variabile viene indicato dal valore assunto da un array indicizzato con l'identificatore del processo. Ad esempio, la variabile $\texttt{state}_p$ dello pseudocodice in Algoritmo 5.2.1 viene indicata con $\texttt{state}[p]$ nella formalizzazione. Nelle prossime sezioni verranno analizzate in dettaglio tutte le transizoni del sistema.

Inizializzazione

All'inizio del protocollo il sistema si trova nel primo round, nessun processo è deciso, né è coordinatore e tanto meno lo è mai stato. Nessun processo ha già eseguito le operazioni del primo round e non è ancora stata ricevuta alcuna *request*. La $\forall^I$-formula che descrive l'insieme degli stati iniziali del sistema è quindi

$$\forall x \left(\begin{array}{l} (\texttt{round}[x] = 1) \wedge (\texttt{state}[x] = \texttt{false}) \wedge \\ (\texttt{coord}[x] = \texttt{false}) \wedge (\texttt{aCoord}[x] = \texttt{false}) \wedge \\ (\texttt{done}[x] = \texttt{false}) \wedge (\texttt{request}[x] = \texttt{false}) \end{array} \right) \tag{5.1}$$

Utilizzando la sintassi accettata dal traduttore descritto nel Capitolo 3 è possibile esprimere la configurazione iniziale del sistema come:

```
initial (universal x:nat) {
  (round[x] = 1) AND (state[x] = false) AND (coord[x] = false) AND
  (aCoord[x] = false) AND (done[x] = false) AND (request[x] = false)
}
```

Stati finali

Gli stati finali rappresentano una particolare configurazione del sistema; scopo della verifica è, in questo caso, provare formalmente che il sistema non può raggiungere in nessuna possibile esecuzione uno di questi stati. Come riportato nel Capitolo 2, l'insieme degli stati finali viene espresso con una $\exists^I$-formula. Tale formula, in questo caso, deve rappresentare una configurazione nella quale non sia soddisfatta la proprietà di *agreement* espressa nella Definizione 4.2.1, ovvero deve evidenziare il fatto che vi sono almeno due processi del sistema che hanno deciso, sono corretti e il loro valore di decisione è diverso.

Il modello di fallimento *stopping failure* assunto da MCMT permette di evitare l'introduzione di una nuova variabile per indicare quali siano i processi errati. L'assenza di questa variabile da un lato riduce il numero di transizioni da specificare per modellare il protocollo, ma dall'altro non permette la formalizzazione della proprietà di *agreement*, che è la proprietà di interesse per il problema di *safety* oggetto di verifica. Infatti la formula degli stati finali sarebbe dovuta essere questa:

$$\exists x, y \left(\begin{array}{l} x \neq y \wedge \\ \texttt{state}[x] = \texttt{true} \wedge \texttt{faulty}[x] = \texttt{false} \wedge \\ \texttt{state}[y] = \texttt{true} \wedge \texttt{faulty}[y] = \texttt{false} \wedge \\ \texttt{decisionValue}[x] \neq \texttt{decisionValue}[y] \end{array} \right) \tag{5.2}$$

assumendo che la variabile `faulty` valga `true` per i processi che falliscono.

L'assenza della variabile `faulty`, tuttavia, permette di esprimere un'altra proprietà del sistema, la proprietà di *uniform agreement*, indicata dalla formula

$$\exists x, y \left(\begin{array}{l} x \neq y \wedge \\ \texttt{state}[x] = \texttt{true} \wedge \texttt{state}[y] = \texttt{true} \wedge \\ \texttt{decisionValue}[x] \neq \texttt{decisionValue}[y] \end{array} \right) \tag{5.3}$$

L'insieme di stati rappresentati dalla (5.2) è un sottoinsieme proprio dell'insieme di stati rappresentato dalla (5.3). Per questo problema di *safety* verrà quindi assunta come configurazione degli stati finali la (5.3), che può essere espressa secondo la sintassi accettata dal traduttore del Capitolo 3 come

```
unsafe (existential x:nat, existential y:nat) {
  (state[x] = true) AND (state[y] = true) AND
  (decisionValue[x] != decisionValue[y])
}
```

Primo Round

Durante il primo round i processi indecisi inviano al coordinatore una *request* (linea 7, Algoritmo 5.2.1).

Dato che il sistema distribuito in esame può essere rappresentato da n processi identificati univocamente da un *id* nell'intervallo $[1, n]$, che possono comunicare attraverso collegamenti affidabili in una rete *point-to-point* completamente connessa, l'invio di un messaggio di *request*[1] equivale alla modifica della variabile condivisa globale `request`.

Il comportamento di ogni processo p è determinato dal valore della variabile `state`$[p]$: se il processo non è deciso (ovvero vale `state`$[p]$ = `false`) invierà al coordinatore un messaggio di *request*, mentre se ha già deciso non farà nulla in questo round. L'altra variabile che determina il numero delle transizioni per questo round è `coord`: dato che MCMT assume implicitamente che le variabili esistenzialmente quantificate delle formule che rappresentano le transizioni rappresentino due processi distinti, non è possibile modellare con un unica transizione l'invio del messaggio di *request*. Servirà una transizione per modellare l'invio della *request* effettuato da un processo p al coordinatore q, dove $p \neq q$, ed un'altra per modellare un coordinatore indeciso che si auto-invia quel messaggio. I processi decisi invece non eseguono alcuna operazione indipendentemente dal valore della variabile `coord`, quindi non c'è bisogno di definire due diverse transizioni per modellarne il comportamento.

Per questi motivi, nella formalizzazione finale vi saranno tre transizioni relative al primo round. Nella guardia di ognuna di esse verranno menzionate, oltre a `coord` e `state`, le variabili `round`, che dovrà assumere necessariamente il valore 1 e `done`, che dovrà valere `false`, indicando così che il processo in questione non ha ancora svolto le operazioni del round. Ricordando il formato

[1] Questo discorso può essere esteso all'invio di qualsiasi altro messaggio in rete: inviare un messaggio equivale a modificare una variabile condivisa.

delle transizioni indicato nel Capitolo 2, le transizioni di questo round saranno le seguenti[2]:

- il sistema si trova nel primo round ($\texttt{round}[x] = 1$, e questo letterale assume lo stesso valore di verità indipendente dal valore di x), c'è un processo x indeciso ($\texttt{state}[x] = \texttt{false}$) che non ha ancora svolto le operazioni del round ($\texttt{done}[x] = \texttt{false}$) ed il processo y è coordinatore ($\texttt{coord}[y] = \texttt{true}$). Allora il processo x invierà un messaggio di *request* al coordinatore ($\texttt{request}' = \lambda j.\texttt{true}$) e la variabile done verrà impostata a true per indicare che questo processo ha già eseguito le operazioni del round ($\texttt{done}' = \lambda j.(\textsf{if } x = j \textsf{ then } \texttt{true} \textsf{ else } \texttt{done}[j])$):

$$\tau_1 := \exists x, y \left(\begin{array}{l} (x \neq y) \wedge (\texttt{round}[x] = 1) \wedge (\texttt{done}[x] = \texttt{false}) \wedge \\ (\texttt{state}[x] = \texttt{false}) \wedge (\texttt{coord}[y] = \texttt{true}) \wedge \\ \texttt{request}' = \lambda j.\texttt{true} \wedge \\ \texttt{done}' = \lambda j.(\textsf{if } x = j \textsf{ then } \texttt{true} \textsf{ else } \texttt{done}[j]) \end{array} \right) \tag{5.4}$$

- il sistema si trova nel primo round ($\texttt{round}[x] = 1$, indipendentemente dal valore di x), c'è un processo x indeciso ($\texttt{state}[x] = \texttt{false}$) che non ha ancora svolto le operazioni del round ($\texttt{done}[x] = \texttt{false}$) e che ricopre la carica di coordinatore ($\texttt{coord}[x] = \texttt{true}$). Allora il processo x invierà un messaggio di *request* a se stesso ($\texttt{request}' = \lambda j.\texttt{true}$) e la variabile done verrà impostata a true per indicare che questo processo ha già eseguito le operazioni del round ($\texttt{done}' = \lambda j.(\textsf{if } x = j \textsf{ then } \texttt{true} \textsf{ else } \texttt{done}[j])$):

$$\tau_2 := \exists x \left(\begin{array}{l} (\texttt{round}[x] = 1) \wedge (\texttt{done}[x] = \texttt{false}) \wedge \\ (\texttt{state}[x] = \texttt{false}) \wedge (\texttt{coord}[x] = \texttt{true}) \wedge \\ \texttt{request}' = \lambda j.(\texttt{true}) \wedge \\ \texttt{done}' = \lambda j.(\textsf{if } x = j \textsf{ then } \texttt{true} \textsf{ else } \texttt{done}[j]) \end{array} \right) \tag{5.5}$$

[2]In questo paragrafo sono state riportate le transizioni espresse sia secondo la forma descritta nel Capitolo 2, sia secondo il formalismo accettato dal traduttore descritto nel Capitolo 3. Dato che le due forme sono molto simili, in futuro la prima verrà omessa e verrà riportata solo la seconda.

- il sistema si trova nel primo round ($\texttt{round}[x] = 1$, indipendentemente dal valore di x), c'è un processo x deciso ($\texttt{state}[x] = \texttt{true}$) che non ha ancora svolto le operazioni del round ($\texttt{done}[x] = \texttt{false}$). Questo processo non ha operazioni da svolgere in questo round, quindi la variabile `done` verrà impostata a `true` per indicare che questo processo è pronto per passare al round successivo ($\texttt{done}' = \lambda j.(\textsf{if } x = j \textsf{ then } \texttt{true} \textsf{ else } \texttt{done}[j])$):

$$\tau_3 := \exists x \left(\begin{array}{l} (\texttt{round}[x] = 1) \wedge (\texttt{done}[x] = \texttt{false}) \wedge (\texttt{state}[x] = \texttt{true}) \wedge \\ \texttt{done}' = \lambda j.(\textsf{if } x = j \textsf{ then } \texttt{true} \textsf{ else } \texttt{done}[j]) \end{array} \right) \tag{5.6}$$

Queste transizioni possono essere espresse secondo il formalismo accettato dal traduttore descritto nel Capitolo 3:

```
/* Transition 1 */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND
         (state[x] = false) AND (coord[y] = true)
  update:
    request := true;
    done[x] := true;
}
/* Transition 2 */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (coord[x] = true) AND
         (done[x] = false) AND (state[x] = false)
  update:
    request := true;
    done[x] := true;
}
/* Transition 3 */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = true)
  update:
    done[x] := true;
}
```

Passaggio dal round 1 al round 2

Per formalizzare il passaggio dal round 1 al round 2 (linee 8-10, Algoritmo 5.2.1) bisogna tener presente le due possibili configurazioni nelle quali si può trovare il sistema. Nella prima il coordinatore ha ricevuto almeno un messaggio di *request*, quindi il sistema passerà al secondo round. Nella seconda il coordinatore non riceve alcuna *request*, quindi deve essere dimesso ed i round 2 e 3 non dovranno essere eseguiti.

Le transizioni saranno dunque due, nella guardia di una verrà indicato `request`$[x]$ = `true` (ricordiamo che `request` è una variabile globale, quindi assume lo stesso valore indipendentemente da x), mentre nell'altra `request`$[x]$ = `false`. Inoltre, in entrambe le transizioni bisognerà indicare che il sistema si trova nel primo round e che tutti i processi hanno eseguito le operazioni del round. Nella guardia "esistenziale" (una formula nella quale compaiono solo variabili esistenzialmente quantificate, indicata nel linguaggio del traduttore con la parola riservata `guard:`) viene menzionato anche che deve esistere il coordinatore, ovvero vi è il letterale `coord`$[x]$ = `true`. Questo perché se il coordinatore fallisce il sistema non deve passare al round successivo, bensí deve essere eletto un nuovo coordinatore. Quest'ultima operazione di elezione verrà discussa più avanti; per ora è sufficiente sottolineare che il passaggio di round viene eseguito solo se esiste un processo non fallito che è coordinatore. Dato che MCMT assume implicitamente la disgiunzione delle variabili che rappresentano i processi, nella guardia esistenziale verrà indicato che anche il coordinatore deve aver eseguito le operazioni del round (`done`$[x]$ = `true`); senza questo letterale la transizione non sarebbe corretta in quanto non verrebbe controllato se il coordinatore ha eseguito le operazioni del round.

Il passaggio di round, che avviene se `request`$[x]$ = `true` è modellato da questa transizione (la numerazione segue dalle transizioni precedenti):

```
/* Transition 4 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = true) AND
         (coord[x] = true) AND (done[x] = true)
  uguard: (done[all] = true)
  update:
    round := 2;
```

```
    done := lambda (j:nat) { false }
}
```

mentre la seconda situazione da prevedere, il caso nel quale `request`$[x]$ = `true`, è modellata da quest'altra transizione:

```
/* Transition 5 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = false) AND
         (coord[x] = true) AND (done[x] = true)
  uguard: (done[all] = true)
  update:
    coord[x] := false;
    aCoord[x] := true;
}
```

Secondo Round

Nel secondo round (linee 10-13 Algoritmo 5.2.1) il coordinatore deve inviare ai processi indecisi la propria stima. Anche in questo round il comportamento dei processi è determinato dal valore delle variabili `state` e `coord`: sebbene il coordinatore invii in modalità *broadcast* (ossia a tutti i processi del sistema) la propria stima, i processi p decisi (per i quali vale `state`$[p]$ = `true`) la ignorano, quindi a livello formale non fanno nulla in questo round. Invece i processi indecisi ricevono la stima del coordinatore e la salvano come propria stima (linea 12, Algoritmo 5.2.1). Le transizioni sono quindi tre: le prime due modellano il comportamento dei processi indecisi (in una coordinatore e ricevente della stima sono due processi distinti, mentre nell'altra sono lo stesso processo) mentre la terza riguarda i processi decisi.

Bisogna fare attenzione al fatto che il letterale `done`$[p]$ = `false` nella guardia indica che il processo p non ha ancora ricevuto la stima. La variabile `done`, in questo round, non indica se un processo ha già eseguito o meno le operazioni previste per esso (questa cosa non avrebbe senso dato che l'unico vero attore di questo round è il coordinatore!) ma indica semplicemente se ha già ricevuto o meno la stima del coordinatore. In altre parole questa variabile può essere pensata come un *flag* utilizzato dal coordinatore per sapere quali sono i processi ai quali ha già inviato la propria stima (ricordiamo che il

sistema distribuito in esame non supporta primitive per il *broadcast* quindi inviare un messaggio in tale modalità implica l'apertura di n connessioni point-to-point, dove n è la dimensione del sistema).

Ecco le tre transizioni del round:

```
/* Transition 6 */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND
         (state[x] = false) AND (coord[y] = true)
  update:
    estimate[x] := estimate[y];
    done[x] := true;
}
/* Transition 7 */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (state[x] = false) AND
         (done[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
}
/* Transition 8 */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (state[x] = true) AND (done[x] = false)
  update:
    done[x] := true;
}
```

La transizione 7 potrebbe far sorgere qualche dubbio: essa dovrebbe modellare un coordinatore indeciso che si auto-invia il messaggio di *request*, quindi la parte di *update* avrebbe dovuto avere l'istruzione che salva nella variabile `estimate` la nuova stima ricevuta. Però, dato che è il coordinatore stesso che si auto-invia questo messaggio, l'istruzione di update sarebbe stata `estimate[x] := estimate[x]`, e quindi può essere omessa perché non ha effetto sul valore della variabile.

Passaggio dal round 2 al round 3

Il passaggio dal secondo round al terzo viene effettuato se vi è un coordinatore non fallito nel sistema e se tutti i processi hanno eseguito le operazioni

del secondo round. La transizione che modella questo comportamento è la seguente:

```
/* Transition 9 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 2) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 3;
    done := lambda (j:nat) { false }
}
```

Terzo Round

Durante il terzo round (linee 14-19, Algoritmo 5.2.1) il coordinatore invia in modalità *broadcast* il messaggio *decide*; tutti i processi che ricevono questo messaggio eseguono le linee 17 e 18 dello pseudocodice riportato in Algoritmo 5.2.1. Di fatto, semanticamente, queste due operazioni indicano la consegna di un messaggio da parte del processo del livello *Reliable Broadcast* all'applicazione.

Come per i due round precedenti, il comportamento dei processi è definito dal contenuto delle variabili `state` e `coord`, e se per qualche processo x vale `state`$[x]$ = `true` le operazioni da eseguire sono le stesse indipendentemente dal valore `coord`$[x]$. Ecco le transizioni:

```
/* Transition 10 */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND
         (state[x] = false) AND (coord[y] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* Transition 11 */
transition (existential x:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND
         (state[x] = false) AND (coord[x] = true)
  update:
    state[x] := true;
```

```
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* Transition 12 */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (state[x] = true) AND (done[x] = false)
  update:
    done[x] := true;
}
```

Elezione del coordinatore

I tre round dell'algoritmo vengono eseguiti ciclicamente, per $t + 1$ volte,[3] ed ogni volta che si riparte dal primo round viene eletto un nuovo processo che non è mai stato prima coordinatore come tale, e viene "dimesso" quello in carica.

Per modellare questo comportamento è stato introdotto un nuovo round. Quando termina il terzo round il sistema passa ad un quarto dove vengono eseguite queste nuove operazioni. Ecco la transizione che porta il sistema nel round aggiunto:

```
/* Transition 13 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 3) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 4;
    done := lambda (j:nat) { false }
}
```

Come gli altri passaggi di round, l'unica condizione necessaria perché il sistema passi al round successivo - oltre al fatto che per ogni processo x del sistema deve valere `done`$[x] =$ `true` - è che il processo coordinatore sia ancora in esecuzione e non sia fallito.

Una volta che il sistema è nel quarto round deve essere dimesso il coordinatore in carica e deve essere eletto come nuovo coordinatore un processo

[3] Il parametro t indica il numero di processi che falliscono nel sistema. Si dimostra infatti che [13] è sufficiente 1 processo corretto nella rete perché venga soddisfatta la proprietà di *agreement*.

che non ha mai assunto questo ruolo. Nella guardia della transizione devono apparire due variabili esistenzialmente quantificate - siano, ad esempio, x ed y - che rappresentino il coordinatore attualmente in carica ed il coordinatore che verrà eletto. Per il processo y che verrà eletto come nuovo coordinatore deve valere `aCoord`$[y]$ = `true`, ed ovviamente `coord`$[y]$ = `false`. Nella sezione di update della transizione bisogna ripristinare il valore iniziale della variabile `request` e riportare il sistema nel primo round. Ecco la transizione:

```
/* Transition 14 */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 4) AND (coord[x] = true) AND
         (coord[y] = false) AND (aCoord[y] = false)
  update:
    round := 1;
    coord[x] := false;
    aCoord[x] := true;
    done := lambda (j:nat) { false }
    request := false;
    lambda (j:nat; x != j) {
      case y = j:
        coord[j] := true;
      otherwise:
        coord[j] := coord[j];
    }
}
```

Bisogna ricordarsi anche di inserire una transizione che viene applicata quando il coordinatore fallisce. In questo caso, nel sistema vale `coord`$[x]$ = `false` per qualsiasi processo x: è questa l'unica condizione da porre nella guardia della transizione che serve per far ripartire il sistema dal primo round con un nuovo coordinatore (dato che il sistema riparte dal primo round bisogna anche ripristinare il valore di default della variabile `request`). Ecco l'ultima transizione della formalizzazione:

```
/* Transition 15 */
transition (existential x:nat, universal all:nat) {
  guard: (coord[x] = false) AND (aCoord[x] = false)
  uguard: (coord[all] = false)
  update:
    round := 1;
```

```
        coord[x] := true;
        done := lambda (j:nat) { false }
        request := false;
    }
```

Ottimizzazioni

Come si è visto, il comportamento di un processo è determinato in tutti i round dal valore delle variabili `state` e `coord`. In base al valore della prima il processo è "attivo" o "passivo" all'interno del sistema: se per un processo p vale `state`$[p]$ = `false`, questo processo sarà "attivo", ovvero invierà messaggi di *request*, riceverà la stima dal coordinatore e successivamente la *decide*. In caso contrario il processo non svolgerà alcuna operazione. L'altra variabile che determina il comportamento dei processi è `coord`: se `coord`$[p]$ = `true` il processo dovrà ricevere i messaggi di *request* e in base a quanti ne riceve eseguire o meno i due round successivi. Le transizioni per formalizzare il passaggio di round in round sono dunque nove (in caso che il processo è deciso non serve sapere se sia coordinatore o meno: esegue sempre la stessa operazione).

In aggiunta a queste, bisogna inserire altre transizioni per modellare

- l'elezione di un nuovo coordinatore nel caso in cui quello in carica fallisce;
- l'elezione di un nuovo coordinatore al termine del ciclo `for` più esterno (linea 5);
- il passaggio da un round all'altro (due per i round "ordinari" - dal primo al secondo e dal secondo al terzo - più una per permettere il passaggio dal terzo al quarto round nel quale viene eletto il coordinatore)

Inoltre bisogna ricordare che il passaggio dal primo round al secondo avviene solo se è stato ricevuto almeno un messaggio di *request* (linea 8); per questo comportamento bisogna aggiungere una transizione, per un totale di quindici.

Questo numero però può essere ridotto: è possibile notare che il comportamento dei processi decisi (che in pratica non fanno nulla) è indipendente dal round nel quale si trova il sistema e dal contenuto di `coord`, quindi le tre

transizioni che lo modellano (una per ogni round) possono essere accorpate in una sola, omettendo di indicare nella guardia il valore del round. In totale le transizioni sono tredici, e sono riportate in Appendice A.1.

5.3 Verifica

La verifica del protocollo ha richiesto $1,176$ secondi. MCMT è stato eseguito solo con l'opzione `-S3`. Altri dati relativi alla verifica:

- Numero di variabili nella formalizzazione: 8;
- Numero di transizioni nella formalizzazione: 13;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 4;
- profondità massima raggiunta: 13;
- nodi generati: 113;
- nodi cancellati: 21;
- chiamate all'SMT-solver: 2.792;

6

Dal modello di fallimento Crash al modello Send-Omission

Il modello di fallimento successivo (in grado di difficoltà) a quello *crash* è il modello detto "*omission*", che può essere diviso in *receive-omission* o *send-omission* a seconda che i processi falliscano in ricezione o durante l'invio di messaggi. Mentre un fallimento in ricezione è facile da rilevare da parte del processo che sta fallendo (i tempi di invio e ricezione sono noti e i processi sono sincronizzati, quindi se p non riceve un messaggio l'errore è stato commesso da esso stesso), la creazione e la verifica di un protocollo per garantire l'*agreement* in caso di fallimenti in *send-omission* sono compiti più complicati.

Il protocollo proposto in [13] per risolvere il problema del *Reliable Broadcast* in caso di fallimenti *send-omission* è "costruito" a partire da quello per i fallimenti *crash*. Gli autori mostrano quali sono le modifiche da effettuare al protocollo riportato in Algoritmo 5.2.1 per migliorarlo e potenziarlo, affinché riesca a mantere la consistenza delle decisioni almeno dei processi corretti in caso di fallimenti *send-omission*. Seguendo il percorso tracciato sull'articolo di riferimento, nella prima parte del capitolo mostrerò (sempre utilizzando MCMT) che il protocollo riportato in Algoritmo 5.2.1 non riesce a risolvere il *Reliable Broadcast* con fallimenti *send-omission*. Dato che MCMT riporta, in caso di fallimento, una sequenza di transizioni (detta anche traccia) che a partire da uno stato iniziale porta il sistema ad uno stato finale *unsafe*,

analizzerò questa traccia, dando un esempio d'esecuzione che porta all'inconsistenza della decisione tra *almeno* due processi corretti. Nella seconda parte del capitolo analizzerò una prima variante del protocollo riportato in Algoritmo 5.2.1, individuando anche per questa versione una possibile esecuzione che permette di raggiungere la configurazione *unsafe*.

6.1 Modello di fallimento

Questo nuovo modello di fallimento è descritto dalle seguenti definizioni:

Definizione 6.1.1. Secondo il modello di fallimento *send-omission*, un processo è considerato *errato* se omette di inviare almeno un messaggio che avrebbe dovuto inviare.

Un processo errato secondo la Definizione 5.1.1 è errato anche per la Definizione 6.1.1: esso può essere inteso come un processo che da un certo punto in poi non invia più messaggi. Tuttavia non è vero il contrario: un processo errato secondo la Definizione 6.1.1 è un processo che può inviare messaggi "ad intermittenza", come se ci fosse un cavo di rete difettoso. Allora, secondo questo modello di fallimento vale la seguente

Definizione 6.1.2. Un processo è *corretto* se invia tutti i messaggi correttamente.

6.2 Formalizzazione

Per modellare il comportamento del protocollo riportato in Algoritmo 5.2.1 assumendo il modello di fallimento *Send-Omission* è necessario introdurre nel sistema una variabile locale `faulty`, dove `faulty`$[p]$ = `true` indica che il processo p è errato secondo la Definizione 6.1.1. Infatti il modello `stopping failure` assunto implicitamente dal sistema non è più sufficiente per catturare la nuova definizione di processo errato. La variabile `faulty` serve proprio per estendere la formalizzazione di processo errato: il sistema continuerà a gestire i fallimenti *crash* automaticamente, e come descritto nel Capitolo 2, ogni variabile quantificata viene relativizzata rispetto al fatto

che il processo rappresentato da essa deve essere ancora in esecuzione. Con la variabile `faulty`, invece, vengono modellati i fallimenti esclusivamente in *send-omission*, ossia quei fallimenti la cui caratteristica fondamentale è l'occasionalità nel l'invio dei messaggi, e che rendono il modello di fallimento attuale più severo del precedente.

Il cambio di modello di fallimento implica necessariamente un aumento di transizioni: l'invio di un messaggio da parte di un processo può fallire o meno come nel modello precedente, ma ora il processo dopo il fallimento potrebbe essere ancora in esecuzione!

Bisogna intervenire in tre diversi punti: nel primo round i processi inviano un messaggio di *request* al coordinatore, ma potrebbero anche fallire. Questo comportamento deve essere modellato da una nuova transizione:

```
/* A faulty undecided process fails sending the request message */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND
         (state[x] = false) AND (coord[y] = true)
  update:
    faulty[x] := true;
    done[x] := true;
}
```

Stesso discorso vale per il coordinatore che deve inviare la propria stima (nel secondo round) e il messaggio *decide* (nel terzo round): l'invio di questi messaggi potrebbe fallire in qualche caso, quindi alla formalizzazione vanno aggiunte queste due transizioni:

```
/* A faulty coordinator fails sending his estimate */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND
         (state[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; x != j) {
      case y = j :
        faulty[j] := true;
      otherwise :
        faulty[j] := faulty[j];
    }
```

```
}
/* The coordinator fails sending a 'decide' */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND
         (state[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; x != j) {
      case (j = y): faulty[j] := true;
      otherwise: faulty[j] := faulty[j];
    }
}
```

In definitiva, l'aggiunta della nuova variabile `faulty` implica un aumento di transizioni, in quanto il comportamento di ogni processo è determinato, oltre che dal valore di `state` e `coord`, anche da questa nuova variabile. Tuttavia si suppone che un coordinatore non possa fallire quando invia i messaggi a se stesso, quindi di fatto le transizioni in più sono solo le tre appena descritte.

In totale le transizioni richieste per modellare il comportamento del sistema sono sedici.

6.2.1 Analisi della traccia unsafe

Il protocollo riportato in Algoritmo 5.2.1 non riesce a risolvere il problema del Reliable Broadcast se il sistema ammette anche fallimenti *send-omission*. MCMT, dopo $17,657$ secondi indica che il sistema è *unsafe*,[1] mostrando una

[1]Il tool è stato eseguito senza opzioni. Altri dati relativi all'esecuzione sono:

- Numero di variabili nella formalizzazione: 9;
- Numero di transizioni nella formalizzazione: 16;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 5;
- massima profondità raggiunta: 12;
- numero di nodi generati: 464;
- nodi cancellati: 26;
- chiamate all'SMT-solver: 19.733;

sequenza di 11 transizioni che rappresentano l'esecuzione riportata in Figura 6.1.

È interessante confrontare l'esecuzione "estratta" dalla sequenza di transizioni restituite da MCMT come prova del fallimento del protocollo riportato in Algoritmo 5.2.1 assumendo fallimenti *Send-Omission* con quella riportata su [13] per mostrare la "debolezza" del protocollo.

Chandra e Toueg su [13] scrivono:

> [Algorithm 5.2.1] does not tolerate send-omission failures. For example a faulty coordinator c could first omit to send $\texttt{estimate}_c$ to the next coordinator, and then send *decide* to one correct process p. Thus p decides on $\texttt{estimate}_c$ while the next coordinator, unaware of this estimate, can make undecided processes decide on $\perp$. This lead to disagreement.

Questa esecuzione effettivamente porta il sistema in uno stato *unsafe*, tuttavia essa prevede la cooperazione di quattro processi ed un certo numero di transizioni. L'esecuzione ricavata dalla "traccia *unsafe*" restituita da MCMT (riportata in modalità "esplicita in Sezione C.1) invece mostra che l'algoritmo può arrivare in uno stato di disaccordo anche quando i processi che cooperano solo solo 3. Questa traccia, senza dubbio, è più corta rispetto a quella riportata sull'articolo, in quanto se il coordinatore fallisce il protocollo riparte dall'inizio con un nuovo coordinatore, e ciò implica che un'eventuale traccia *unsafe* rappresentante l'esecuzione descritta dagli autori è sicuramente più lunga di quella trovata da MCMT, se non altro perché dovrà includere le transizioni necessarie per l'elezione del nuovo coordinatore e per arrivare fino al terzo round, nel quale potranno essere rintracciati due processi che hanno deciso su due valori diversi.

6.3 Prima variante: inserimento di un *NACK*

La prima variante proposta dagli autori di [13] al protocollo analizzato fin'ora è l'aggiunta di un messaggio di *negative-acknowledge* (*NACK*) che viene inviato al coordinatore da tutti i processi che nel round 1 hanno inviato un messaggio di *request*, ma che nel secondo round non hanno ricevuto alcuna

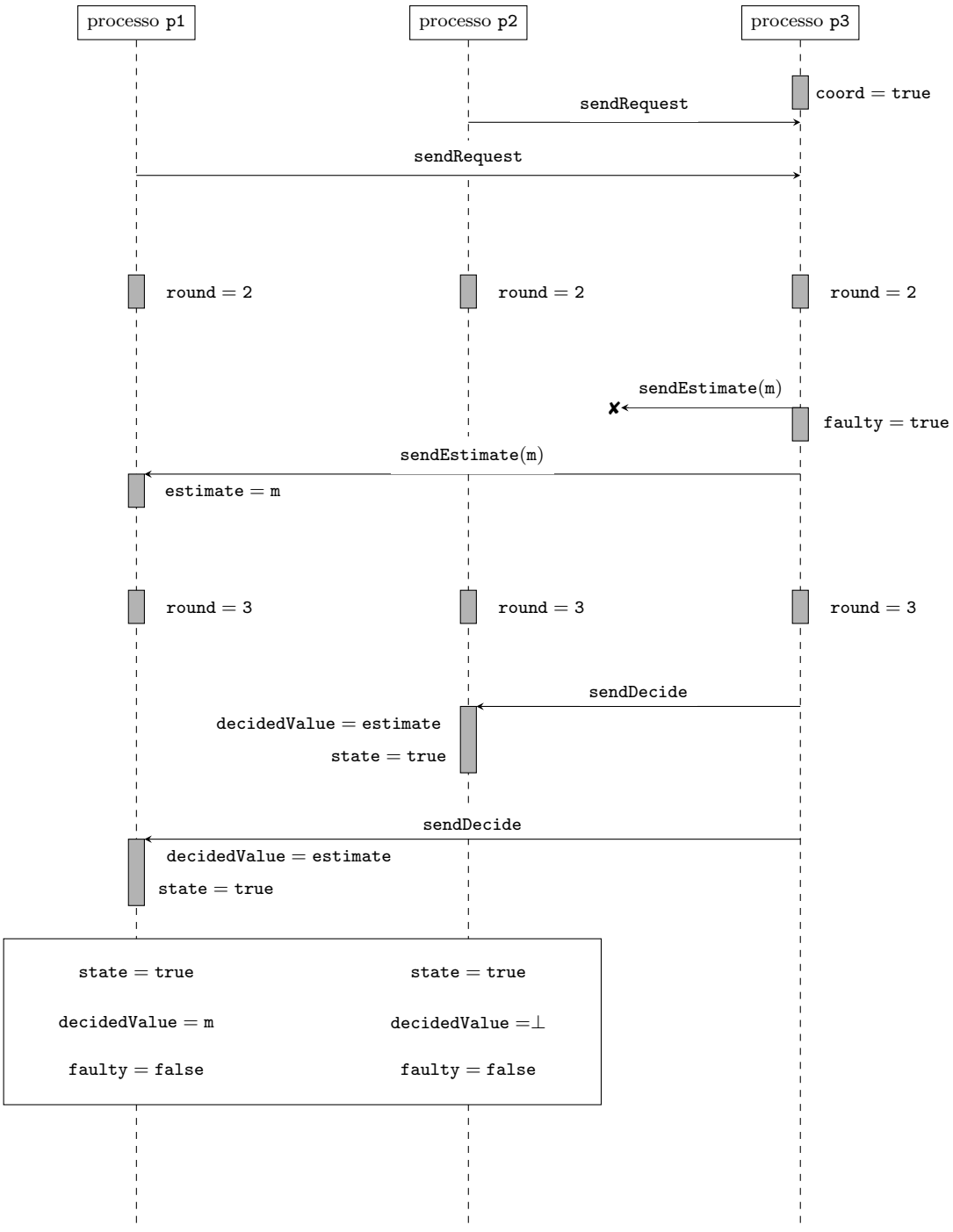

Figura 6.1: Diagramma di sequenza ottenuto dalla "*traccia unsafe*" esibita da MCMT come prova del fallimento dell'Algoritmo 5.2.1 eseguito su un sistema con fallimenti in *send-omission*.

stima. Come si vede nel diagramma in Figura 6.1, i processi `p1` e `p2` raggiungono lo stato di inconsistenza perché il coordinatore invia la sua stima solo al processo `p1`, mentre fallisce durante l'invio della stessa a `p2`. Il nuovo protocollo indica ai processi corretti ed indecisi che non ricevono la stima del coordinatore nel secondo round (come nel caso di `p2` nell'esecuzione di Figura 6.1) di inviare un messaggio di *negative-acknowledge* al coordinatore. Il coordinatore che riceve uno o più messaggi di *negative-acknowledge* si ferma. Il nuovo protocollo è riportato in Algoritmo 6.3.1.

6.3.1 Formalizzazione

Per formalizzare le specifiche riportate in Algoritmo 6.3.1 ho aggiunto alla formalizzazione precedente due nuove variabili locali booleane: `received` e `nack`. La variabile `received` serve per tenere traccia dei processi che hanno ricevuto la stima nel secondo round.

6.3.2 Variabile `received` e relative modifiche

Nelle due transizioni che modellano la ricezione della stima è stata aggiunta, nella sezione di *update*, una nuova istruzione per porre la variabile `received` a `true`:

```
/* 1) An undecided process sends a request message to the coordinator */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND
         (state[x] = false) AND (coord[y] = true)
  update:
    request := true;
    done[x] := true;
}

/* 3) An undecided coordinator sends a request to himself */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND
         (state[x] = false) AND (coord[x] = true)
  update:
    request := true;
```

```
        done[x] := true;
    }
```

6.3.3 Variabile `nack` e relative modifiche

La variabile `nack` è stata introdotta per modellare l'invio/ricezione di un messaggio di *negative-acknowledge*.

Il nuovo protocollo (riportato in Algoritmo 6.3.1) prevede un round in più rispetto al precedente (riportato in Algoritmo 5.2.1), e nel nuovo round (linee 14-16, Algoritmo 6.3.1) i processi che non hanno ricevuto la stima nel secondo round inviano un messaggio di *negative acknowledge* al coordinatore. Si può notare fin da subito che, come per i messaggi di *request* del primo round, non interessa il numero di tali messaggi, ma solo che *almeno uno* venga ricevuto dal coordinatore. Tuttavia la variabile `nack` non è globale, ma locale, ed il motivo verrà spiegato tra poco.

Le specifiche in Algoritmo 6.3.1 indicano che, una volta raggiunto il round 3 (il nuovo round del protocollo), i processi che non hanno ricevuto la stima nel round precedente, ovvero quei processi x per i quali vale `received`$[x]$ = `false`, devono inviare un messaggio di *negative acknowledge* al coordinatore:

```
/* 10) If an undecided process didn't received the estimate sends a
   nack to the coordinator */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND
         (done[x] = false) AND (received[x] = false) AND
         (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; j != x) {
      case (j = y):
        nack[j] := true;
      otherwise:
        nack[j] := nack[j];
    }
}
```

Come per ogni operazione di invio di messaggi bisogna modellare anche il fatto che un processo possa fallire durante l'invio:

```
/* 11) A faulty process who didn't received the estimate fails sending
   a nack */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND
         (done[x] = false) AND (received[x] = false) AND
         (coord[y] = true)
  update:
    faulty[x] := true;
    done[x] := true;
}
```

I processi che hanno ricevuto la stima nel round precedente non fanno nulla in questo round:

```
/* 12) If an undecided process received the estimate does nothing */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND
         (done[x] = false) AND (received[x] = true) AND
         (coord[y] = true)
  update:
    done[x] := true;
}
```

Il coordinatore in questo round non fa nulla, in quanto si suppone che esso non fallisca mai in *send-omission* quando si auto-invia i messaggi:

```
/* 13) The coordinator does nothing in this round */
transition (existential x:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND
         (done[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
}
```

Lo pseudocodice riportato in Algoritmo 6.3.1 indica che se un coordinatore riceve almeno un messaggio di *negative acknowledge* rileva il proprio fallimento e si ferma. Per modellare questo comportamento è possibile sfruttare lo stato q_{crash} gestito automaticamente da MCMT: dopo il terzo round ne è stato aggiunto uno *ad hoc* dove viene controllato il valore della variabile `nack`; la transizione che permette di passare al round successivo (il quarto

nello pseudocodice in Algoritmo 6.3.1) presenta la guardia universale

$$\forall \texttt{x}(\texttt{nack}[\texttt{x}] = \texttt{false})$$

In questo modo se il coordinatore ha ricevuto qualche *negative acknowledge* viene forzato al fallimento *crash* dal sistema. Ecco le transizioni che modellano questo comportamento:

```
/* 14) Round 3 completed, system goes to round 4 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 3) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 4;
}
/* 15) If the coordinator didn't received any nack the system goes to
       the 5th round. */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 4) AND (coord[x] = false)
  uguard: (nack[all] = false)
  update:
    round := 5;
    done := lambda (j:nat) { false }
}
```

A questo punto il round 4 dello pseudocodice in Algoritmo 6.3.1 diventa il numero 5, in quanto il quarto round è stato utilizzato nelle due transizioni precedenti. Per riuscire a modellare il comportamento voluto sono necessarie due transizioni e non una sola perché il passaggio dal quarto round al quinto deve essere modellato con una "classica" transizione che controlla la variabile `done` per tutti i processi e che permette il passaggio di round solo se il coordinatore non è fallito in *crash*.

La transizione numero 15 invece è ben diversa: il coordinatore non viene preso in considerazione nella guardia esistenziale, infatti in essa vi è il letterale $\texttt{coord}[x] = \texttt{false}$. E proprio per il fatto che nella guardia esistenziale compare questo letterale, il valore della variabile `nack` in corrispondenza dell'indice del coordinatore verrà preso in considerazione nella guardia universale. Come spiegato nel Capitolo 2, se un processo non soddisfa la guardia universale, il sistema lo forza al fallimento. A questo punto ci si potrebbe

chiedere: la transizione numero 15 è l'unica che permette al sistema di passare dal quarto round al quinto, quindi deve essere eseguita per forza. Tuttavia, questa modalità di gestione delle guardie universali porta al fallimento anche altri processi che non soddisfano la guardia universale, e ciò si discosta dalla semantica del protocollo riportato in Algoritmo 6.3.1! Per risolvere questo dubbio si deve tornare alla sezione di update della transizione numero 10, dove si vede che l'invio di un messaggio di *negative acknowledge* implica l'impostazione di `nack` a `true` solo per quanto riguarda l'indice del coordinatore. In altre parole, se un processo invia un *negative acknowledge* al coordinatore, l'unica posizione dell'array `nack` che diventa `true` è quella indicata dall'identificatore del coordinatore. Da ciò si deduce che l'unico processo che potrebbe non soddisfare la guardia universale della transizione 15, e quindi l'unico che verrebbe forzato al fallimento per riuscire ad applicarla, è il coordinatore, e ciò è conforme alla semantica dell'algoritmo riportato in Algoritmo 6.3.1.

6.3.4 Sommario delle transizioni

In totale le transizioni sono ventidue: oltre alle sedici presenti nella formalizzazione precedente ne sono state aggiunte altre sei per modellare queste situazioni:

1. un processo indeciso che non ha ricevuto la stima nel round 2 invia un *nack* al coordinatore;
2. un processo errato ed indeciso che non ha ricevuto la stima nel round 2 può fallire l'invio del *nack*;
3. un processo indeciso che ha ricevuto la stima nel round 2 non fa nulla;
4. il coordinatore indeciso non invia alcun *nack* perché sicuramente ha ricevuto la stima nel round 2;
5. se il coordinatore non ha ricevuto *nack* il sistema passa al round 4;
6. se il coordinatore ha ricevuto almeno un *nack* viene forzato al fallimento.

6.3.5 Analisi della traccia unsafe

Nonostante sia stato introdotto l'invio del messaggio di *negative-acknowledge*, il nuovo protocollo non riesce a risolvere il problema del *Reliable Broadcast* ammettendo fallimenti in *Send-Omission*. In [13] gli autori riportano che c'è ancora possibilità di disaccordo tra processi decisi:

> ... a faulty coordinator c omits to send $estimate_c$ to a faulty process c' which fails to send a NACK to c. c does not receive any NACKs and thus proceeds to send some *decides*. Then c' becomes the new coordinator without having received $estimate_c$. At this point it is possible that some correct process decide on $estimate_c$ while other correct processes are still undecided and rely on c' for a decision value.

Questa "dimostrazione" di fallimento fatta dagli autori, può essere schematizzata come segue:

0. Per riuscire a trovare un'esecuzione che porti dallo stato iniziale a quello unsafe sono necessari almeno 2 processi *faulty* che diventano coordinatori sequenzialmente. Oltre ad essi ne servono almeno altri due corretti (sui quali sia possibile verificare che decidono su due stime diverse).

1. Il primo coordinatore c fallisce mentre manda la stima ad un altro processo c'.

2. Il processo c' non riesce ad inviare il messaggio di *negative-acknowledge* al coordinatore c.

3. Il coordinatore c riesce a mandare alcuni messaggi di *decide*.

4. c' diventa coordinatore dopo c, ed ha una stima diversa da quella di c.

5. Quando c' diventa coordinatore, è possibile che qualche processo abbia già deciso, mentre altri possono essere ancora in uno stato di indecisione.

6. c' invierà la propria stima a qualche processo indeciso.

7. c' invierà successivamente un messaggio di *decide* a processi ai quali ha inviato la stima nel punto precedente. Ciò porta il sistema in una configurazione di *unsafe*.

MCMT dopo $1.709,93$ secondi d'esecuzione ha riportato come prova del fallimento una traccia di 33 transizioni[2]. Seguendo le transizioni dalla prima all'ultima si viene a costruire l'esecuzione descritta a parole dagli autori (un diagramma di sequenza dell'esecuzione è riportato in Figura 6.2, mentre in Sezione C.2 è riportata un'analisi più approfondita della traccia mostrata da MCMT).

[2]Il tool è stato eseguito senza opzioni. Altri dati relativi all'esecuzione sono:

- Numero di variabili nella formalizzazione: 11;
- Numero di transizioni nella formalizzazione: 22;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 6;
- massima profondità raggiunta: 34;
- numero di nodi generati: 9.679;
- nodi cancellati: 770;
- chiamate all'SMT-solver: 1.338.058;

Algoritmo 6.3.1: Prima modifica di Algoritmo 5.2.1 per cercare di risolvere il problema del *Reliable Broadcast* in caso di fallimenti *Send-Omission*.

1 **Inizializzazione**

2 $\texttt{estimate}_p \leftarrow \begin{cases} \texttt{m} & \text{se } p \text{ è il } \textit{sender} \ (m\text{: valore inizializzato dal sender}), \\ \perp & \text{altrimenti.} \end{cases}$

3 $\texttt{state}_p \leftarrow \texttt{undecided}$;

4

5 **for** $c \leftarrow 1, 2, \ldots, t+1$ **do**

`/* Il processo` c `diventa coordinatore per tre round */`

6 **Round 1**

7 Tutti i processi p *indecisi* inviano un messaggio di *request* a c

8 Se c non riceve alcun messaggio di *request* salta i round 2, 3 e 4

9

10 **Round 2**

11 c invia in modalità broadcast $\texttt{estimate}_c$

12 Tutti i processi p *indecisi* che hanno ricevuto $\texttt{estimate}_c$ impostano $\texttt{estimate}_p \leftarrow \texttt{estimate}_c$

13

14 **Round 3**

15 Tutti i processi p *indecisi* che non hanno ricevuto $\texttt{estimate}_c$ nel Round 2 inviano un `nack` a c

16

17 **Round 4**

18 **if** *c non ha ricevuto alcun messaggio nack* **then**

19 invia in modalità broadcast il messaggio *decide*

20 **else**

21 $q_{crash}(c) = 1$ `/* Il coordinatore rileva il suo fallimento */`

22 Tutti i processi p *indecisi* che hanno ricevuto il messaggio *decide* eseguono:

23 $\texttt{decision}_p \leftarrow \texttt{estimate}_p$

24 $\texttt{state}_p \leftarrow \texttt{decided}_p$

25

26

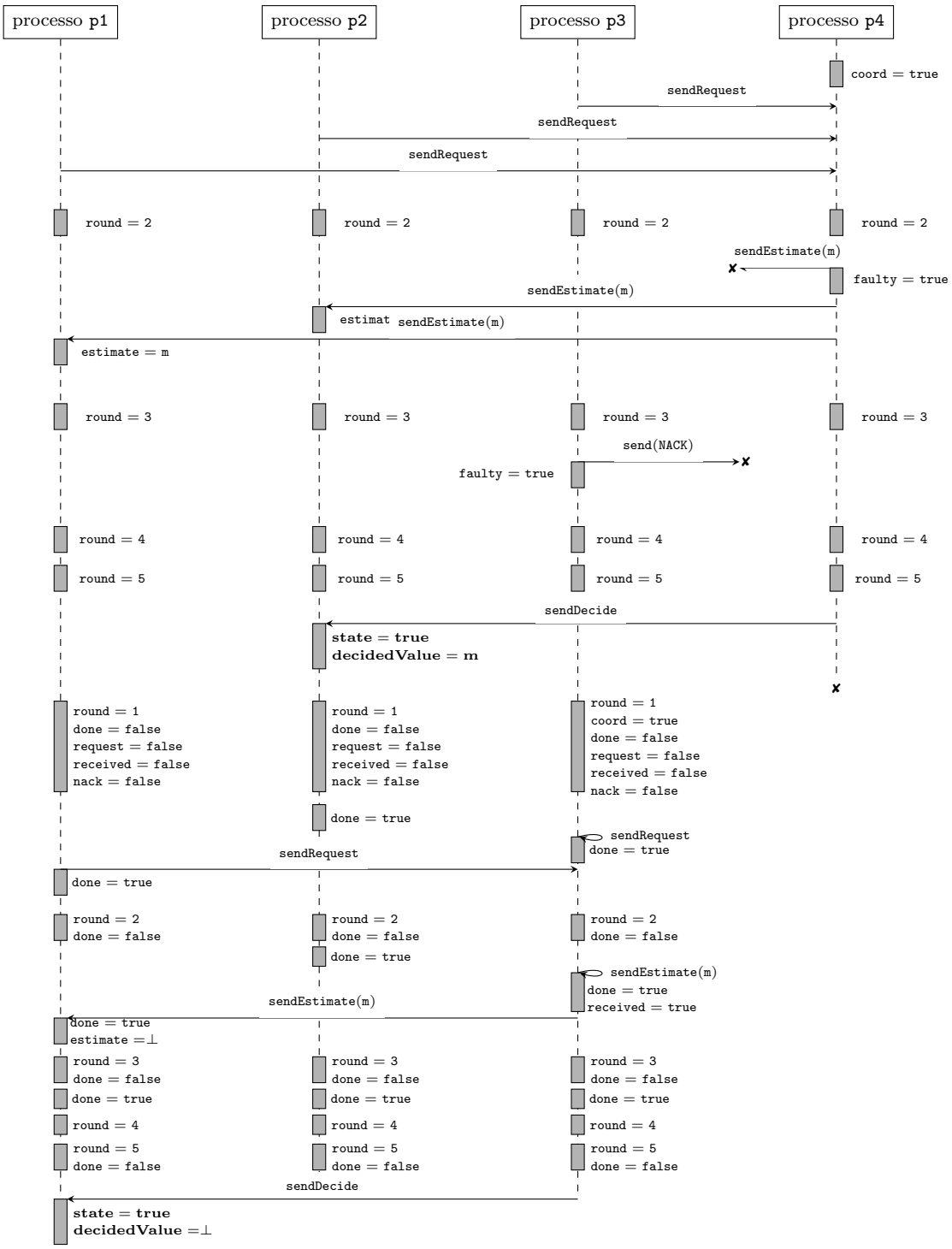

Figura 6.2: Diagramma di sequenza ottenuto dalla "*traccia unsafe*" esibita da MCMT come prova del fallimento dell'Algoritmo 6.3.1 eseguito su un sistema con fallimenti in *send-omission*.

7 Send-Omission Failures

7.1 Algoritmo

L'aggiunta di un messaggio di *negative acknowledge* inviato dai processi al coordinatore per segnalare un mancato invio della stima (Algoritmo 6.3.1) non è sufficiente per risolvere il problema del Reliable Broadcast in caso di fallimenti *Send-Omission*. Bisogna introdurre quindi un'ulteriore modifica: quando un processo invia al coordinatore il messaggio di *request*, allega al messaggio la propria stima e l'identificatore del coordinatore dalla quale l'ha ricevuta. Il coordinatore, dopo aver ricevuto tutte le stime, assumerà come propria quella associata all'identificatore maggiore. Con questa modifica, l'algoritmo è in grado di risolvere il problema del Reliable Broadcast in caso di fallimenti *Send-Omission*. Lo pseudocodice del nuovo algoritmo è riportato in Algoritmo 7.1.1.

Come per i precedenti, la prima parte del capitolo è destinata all'analisi della formalizzazione, mentre la seconda parte verte sulla strategia di verifica adottata per determinare se il protocollo è *safe*.

7.2 Formalizzazione

Per formalizzare il nuovo algoritmo (partendo dal codice riportato in Sezione A.3) è necessario introdurre tre nuove variabili:

- `id` - variabile locale intera che indica l'identificatore del coordinatore dal quale il processo ha ricevuto la stima salvata in `estimate`;
- `maxId` - variabile globale intera utilizzata dal coordinatore per sapere quale sia il maggior identificatore associato alle stime (linea 11);
- `tmpEstimate` - variabile globale utilizzata dal coordinatore per sapere quale sia la stima associata all'identificatore maggiore (linea 11)

Le variabili `maxId` e `tmpEstimate` vengono utilizzate dai processi per inviare la stima al coordinatore: nel primo round ogni processo p confronta il valore $\texttt{id}[p]$ con il contenuto di `maxId`. Se vale $\texttt{id}[p] > \texttt{maxId}$, allora p salva in `maxId` l'identificatore contenuto in $\texttt{id}[p]$ e in `tmpEstimate` la propria stima $\texttt{estimate}_p$. Il primo round (linee 6-13) viene formalizzato in sei transizioni: due sono dedicate ai processi indecisi che riescono ad inviare il messaggio di *request* (una per il caso $\texttt{id}[p] > \texttt{maxId}$ e l'altra per il caso $\texttt{id}[p] \leq \texttt{maxId}$), una transizione formalizza un processo errato che non riesce ad inviare il messaggio di *request*, due per il coordinatore indeciso che si auto invia questo messaggio (anche per il coordinatore c'è la distinzione tra $\texttt{id}[c] > \texttt{maxId}$ e $\texttt{id}[c] \leq \texttt{maxId}$) e l'ultima per i processi già decisi (coordinatore compreso) che non svolgono alcuna operazione. Il passaggio dal primo round al secondo è formalizzato con due transizioni: nella prima il coordinatore ha ricevuto qualche messaggio di *request* mentre nella seconda no, quindi deve essere dimesso. Nel secondo round (linee 14-17) il coordinatore invia la propria stima: una transizione modella l'invio del messaggio $(\texttt{estimate}_c, c)$ ad un processo indeciso (il coordinatore in pratica invierebbe questo messaggio anche ai processi decisi, però questi ultimi lo ignorerebbero, quindi a livello formale questo comportamento è stato tralasciato) mentre un'altra modella il fallimento del coordinatore errato mentre invia il messaggio. Oltre a queste due transizioni ve ne sono altre due: una per il coordinatore indeciso che si autoinvia la stima (di fatto questa operazione consiste nell'impostare $\texttt{id}[c] \leftarrow c$) ed un'ultima per tutti i processi decisi che come sempre non fanno nulla. Dopo la transizione che porta il sistema al terzo round vi sono le transizioni che modellano l'invio dei messaggi di *negative acknowledge* al coordinatore da parte dei processi indecisi che non hanno ricevuto la stima nel primo round (linea 19). Le transizioni sono cinque:

- un processo indeciso che non ha ricevuto la stima invia il messaggio di *negative acknowledge* al coordinatore;
- un processo errato che non ha ricevuto la stima non riesce ad inviare correttamente il messaggio di *negative acknowledge* al coordinatore;
- il coordinatore, i processi che hanno ricevuto la stima nel secondo round ed i processi decisi non fanno nulla in questo round (una transizione per ciascuno);

Il sistema successivamente passa al quarto round (linee 21-29) dove viene forzato al fallimento il coordinatore qualora abbia ricevuto nel round precedente qualche *negative acknowledge* (anche qui come nel protocollo precedente viene utilizzata la guardia universale $\forall$`x`(`nack`[`x`] = `false`) per forzare l'arresto del coordinatore). Viceversa il coordinatore invia il messaggio di *decide* a tutti i processi (se il coordinatore è errato può anche fallire durante l'invio). Nella formalizzazione riportata in Sezione A.4 è stato creato un round successivo al quarto (con indice 5) per queste operazioni. In questo round il coordinatore, se è indeciso, invia un messaggio di *decide* anche a se stesso; i processi già decisi ignorano il messaggio di *decide* eventualmente ricevuto dal coordinatore (quindi formalmente non fanno nulla). Nell'ultimo round introdotto nella formalizzazione (il sesto) viene dimesso il coordinatore in carica, e ne viene eletto uno nuovo. Visto che i coordinatori devono essere eletti in ordine di identificatore non è più possibile dimettere quello in carica ed eleggerne uno nuovo con una sola transizione:[1] nella formalizzazione riportata in Sezione A.4 prima viene dimesso il coordinatore in carica e poi ne viene eletto uno nuovo. L'ultima transizione modella l'elezione di un nuovo coordinatore a seguito del fallimento *crash* di quello in carica. In definitiva la formalizzazione presenta ventotto transizioni:

- sei per il primo round (linee 6-13);
- due per il passaggio dal primo al secondo round (linea 9);

[1] Ciò richiederebbe di avere due variabili quantificate esistenzialmente - il vecchio ed il nuovo coordinatore - ed una guardia universale per controllare che il nuovo coordinatore sia il primo disponibile, ma questa sintassi non è accettata da MCMT

- quattro per il secondo round (linee 14-17);
- una per il passaggio dal secondo al terzo round;
- cinque per il terzo round (linee 18-20);
- una per forzare il fallimento del coordinatore se ha ricevuto messaggi di *negative acknowledge* (linee 22-23) ed una per permettere il passaggio del sistema al quarto round (che nella formalizzazione è indicato come quinto) nel caso non siano stati ricevuti *nack*;
- quattro transizioni per modellare l'invio del messaggio di *decide* (linee 26-28);
- quattro per gestire l'elezione del nuovo coordinatore (anche nel caso in cui quello vecchio sia fallito durante l'esecuzione dell'algoritmo).

7.3 Verifica

La verifica di questo protocollo si è rivelata molto più difficile rispetto ai tre precedenti. MCMT non riesce a determinare[2] se il protocollo formalizzato è *safe* o meno.

A questo punto si aprono due strade (non mutualmente esclusive): cercare degli invarianti per riuscire a potare l'albero di ricerca degli stati raggiungibili a ritroso e sfruttare i tre lemmi riportati su [13] utilizzati dagli autori dell'articolo per dimostrare la correttezza del protocollo.

7.3.1 Ricerca degli invarianti

L'approccio completamente dichiarativo alla base di MCMT è estremamente vantaggioso per la ricerca degli invarianti, in quanto dato un sistema di transizione $\mathcal{S} = (\underline{a}, I, \tau)$ e una formula $\phi(\underline{a})$ che descrive la configurazione che non deve essere raggiunta dal sistema (cioè la configurazione *unsafe*), se MCMT indica che $\mathcal{S}$ è *safe* rispetto alla formula $\phi(\underline{a})$ allora $\neg\phi(\underline{a})$ è un invariante per

[2]Di default MCMT iterrompe l'algoritmo di raggiungibilità all'indietro quando ha generato 50.000 nodi.

$\mathcal{S}$ (per ulteriori informazioni si veda [21]). Questo fatto implica che l'unico "sforzo" lasciato all'utente è la ricerca dei candidati invarianti,[3] mentre la loro verifica può essere eseguita in modalità del tutto automatica con MCMT.

Parallelizzazione della ricerca degli invarianti

Inizialmente, dopo aver appurato che MCMT "da solo" non era in grado provare la *safety* del protocollo nè riusciva a trovare un'esecuzione che portava ad una configurazione *unsafe*, venne realizzato uno script che permetteva di verificare gli invarianti trovati dal tool parallelamente rispetto all'esecuzione dell'algoritmo di raggiungibilità all'indietro, evitando quindi che quest'ultima venisse interrotta. Con la parallelizzazione vennero trovati moltissimi invarianti - circa 250 - tuttavia sperimentalmente si osservò che questi ultimi, invece che facilitare il compito ad MCMT, lo ostacolarono in quanto il tool doveva istanziarli correttamente per tutte le possibili combinazioni di variabili, il che rallentava di molto l'esecuzione dell'algoritmo di raggiungibilità a ritroso. La strategia di ricerca automatica e verifica parallela degli invarianti venne quindi abbandonata, e si passò ad una ricerca manuale - e più mirata - degli stessi.

Invarianti trovati

La formalizzazione attuale presenta sette invarianti:[4]

- non può esserci una configurazione nella quale due processi sono entrambi coordinatori:

$$\forall \mathtt{x}, \mathtt{y} \neg (\mathtt{x} \neq \mathtt{y} \wedge \mathtt{coord}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{coord}[\mathtt{y}] = \mathtt{true})$$

[3] Anche la ricerca "a mano" degli invarianti può essere evitata in quanto MCMT è dotato di alcune euristiche per la loro ricerca attivabili con le opzioni `-iN`, `-a` e `-c`. Tuttavia per i problemi più complessi come quello oggetto di questo capitolo, le euristiche interne di MCMT non sono in grado di trovare invarianti significativi, quindi potrebbe essere richiesta un'interazione con l'utente.

[4] La sintassi di MCMT richiede che gli invarianti suggeriti siano indicati come configurazioni *unsafe*, quindi la negazione delle formule qui riportate.

- se x è coordinatore tutti gli y tali che y < x sono già stati coordinatori:

$$\forall \mathtt{x}, \mathtt{y}\ \neg\,(\mathtt{coord}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{aCoord}[\mathtt{y}] = \mathtt{false} \wedge \mathtt{y} < \mathtt{x})$$

- se x è coordinatore, id[x] non può essere maggiore del suo identificatore x:

$$\forall \mathtt{x}\ \neg\,(\mathtt{coord}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{id}[\mathtt{x}] > \mathtt{x})$$

- per qualsiasi processo y, se x è coordinatore non può accadere che id[y] sia maggiore di x:

$$\forall \mathtt{x}, \mathtt{y}\ \neg\,(\mathtt{x} \neq \mathtt{y} \wedge \mathtt{coord}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{id}[\mathtt{y}] > \mathtt{x})$$

- nel primo round il vincolo precedente è esteso all'uguaglianza: per qualsiasi processo y, se x è coordinatore e vale round[x] = 1 (la variabile round è globale, quindi non cambia valore in funzione dell'indice nel quale viene valutata) id[y] non può essere uguale ad x:

$$\forall \mathtt{x}, \mathtt{y}\ \neg\,(\mathtt{x} \neq \mathtt{y} \wedge \mathtt{round}[\mathtt{x}] = 1 \wedge \mathtt{coord}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{id}[\mathtt{y}] = \mathtt{x})$$

- un processo corretto non riceve mai *negative acknowledge*:

$$\forall \mathtt{x}\ \neg\,(\mathtt{faulty}[\mathtt{x}] = \mathtt{false}) \wedge \mathtt{nack}[\mathtt{x}] = \mathtt{true})$$

- se x è coordinatore per qualsiasi y tale che y ≤ x deve valere aCoord[y] = false:

$$\forall \mathtt{x}, \mathtt{y}\ \neg\,(\mathtt{coord}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{aCoord}[\mathtt{y}] = \mathtt{true} \wedge \mathtt{y} > \mathtt{x})$$

Questi invarianti sono stati inseriti nella formalizzazione utilizzando la direttiva suggested_negated_invariant. In questo modo MCMT li verifica prima di eseguire l'algoritmo di raggiungibilità a ritroso sul problema di safety "completo"; se la verifica di questi invarianti "candidati" determina che sono realmente invarianti, MCMT li utilizzerà come tali. I sette invarianti

trovati richiedono una piccolissima quantità di risorse per essere verificati[5] ma potrebbero dare un contributo enorme durante l'esecuzione dell'algoritmo di raggiungibilità a ritroso! Per capire *quanto* questi invarianti possano essere d'aiuto alla terminazione del problema generale, si consideri il protocollo riportato in Algoritmo 7.1.1 (quindi la formalizzazione riportata in Sezione A.4 togliendo i "suggested invariants" ed i "system axiom"): si vuole verificare che il sistema soddisfa la proprietà di *agreement* assumendo che non esistano processi errati nel sistema (verrà aggiunto nella formalizzazione il codice `system_axiom` (`universal x : nat`) {`faulty`[`x`] = `false`} per indicare questo vincolo). Nella seguente tabella sono riportati i dati delle verifiche con e senza gli invarianti appena descritti:

	Con invarianti	Senza invarianti
Tempo	6,09s	454,59s
Massima profondità	17	26
Nodi generati	91	2.357
Nodi cancellati	5	324
Chiamate all'SMT-solver	7.913	510.436
Invarianti trovati	19 (+7)	0

L'aggiunta degli invarianti diminuisce drasticamente le risorse di tempo e spazio richieste per la terminazione (si passa da poco più di 6 secondi a più di 7 minuti, ed anche i nodi generati passano da 91 a 2.357).

Prima di continuare bisogna fare una precisazione: l'aver mostrato un esempio per il quale questi invarianti danno un grosso contributo alla salvaguardia delle risorse non implica affatto che vi sia un guadagno tangibile anche nella versione completa dell'algoritmo! Di fatto il system axiom introdotto per considerare solo le esecuzioni nelle quali nessun processo sbaglia permette di potare molto l'albero di ricerca degli stati raggiungibili del sistema. Il problema è che non vi è alcuna garanzia che quei sette invarianti

[5]Nel file di specifica l'unica opzione richiesta per far sì che vengano verificati questi invarianti è `(:mcmt :inv_search_max_index_var 4 :)`, la quale indica al tool che possono essere considerati fino a quattro processi distinti per rappresentare gli stati del sistema (di default MCMT utilizza, nella verifica degli invarianti, al massimo tre identificatori per i processi). MCMT impiega 1 secondo circa per verificarli tutti.

portino un beneficio tangibile ed apprezzabile anche nella verifica del problema completo. Di certo un guadagno ci sarà, perché l'albero esplorato in questo esempio è un sottoalbero da esplorare per risolvere il problema completo, tuttavia non si può sapere se (e soprattutto quanto) questi invarianti contribuiranno nell'esplorazione della restante parte dell'albero.

Verifica con i 7 invarianti

Aggiungendo i sette invarianti, MCMT riesce a terminare l'algoritmo di raggiungibilità a ritroso ed indica che il protocollo è `SAFE` in 4.663,74 secondi. Altri dati relativi alla verifica sono:

- Numero di variabili nella formalizzazione: 15;
- Numero di transizioni nella formalizzazione: 28;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 6;
- Massima profondità raggiunta: 39
- Numero di nodi generati: 11.206
- Numero di nodi cancellati: 1.290
- Numero di chiamate all'SMT-solver: 2.563.756
- Numero di invarianti trovati (oltre ai 7 suggeriti): 19

Come sempre MCMT è stato eseguito senza opzioni.

7.3.2 Utilizzo dei tre lemmi

Sull'articolo dal quale è stato tratto il protocollo in analisi ([13]), gli autori danno una dimostrazione di *safety* dello stesso basata su tre lemmi. L'utilizzo di questi tre lemmi in qualità di *system axiom* potrebbe portare ulteriori benefici e permettere una salvaguardia ulteriore di risorse! Ogni lemma verrà formalizzato e quindi verificato utilizzando MCMT. In questo caso si preferisce

verificare i lemmi separatamente (invece di utilizzare la strategia dei "candidati invarianti" mostrata precedentemente con i sette invarianti) in quanto MCMT dedica pochissime risorse alla verifica dei candidati invarianti suggeriti dall'utente. Se queste risorse non dovessero essere sufficienti l'invariante verrebbe scartato e non utilizzato durante l'algoritmo di raggiungibilità a ritroso.

Primo lemma

Lemma 7.3.1. *Sia T il round nel quale viene ricevuto il primo messaggio di decide da qualsiasi processo. Sia p il coordinatore che invia questo messaggio e sia* $\texttt{estimate}_p$ *il messaggio che p ha inviato nel round $T-2$. Alla fine del round $T-2$ tutti i processi q corretti hanno ricevuto* $\texttt{estimate}_p$ *ed hanno impostato* $\texttt{estimate}_q \leftarrow \texttt{estimate}_p$.

Il Lemma 7.3.1 focalizza l'attenzione sul primo messaggio di *decide* ricevuto da un qualsiasi processo. Per esprimere questo invariante nella sintassi accettata da MCMT è necessario introdurre una variabile storica che indichi il processo che ha ricevuto la prima *decide*. Tuttavia, se si trascura questo fatto si ottiene un nuovo lemma che descrive un insieme di esecuzioni per i quali quelle descritte dal Lemma 7.3.1 sono un sottoinsieme:

Lemma 7.3.2. *Sia T il round nel quale viene ricevuto un messaggio di decide da qualsiasi processo. Sia p il coordinatore che invia questo messaggio e sia* $\texttt{estimate}_p$ *il messaggio che p ha inviato nel round $T-2$. Alla fine del round $T-2$ tutti i processi q corretti ed indecisi hanno ricevuto* $\texttt{estimate}_p$ *ed hanno impostato* $\texttt{estimate}_q \leftarrow \texttt{estimate}_p$.

Questo Lemma può essere espresso nella sintassi di MCMT, ed è verificato se il sistema non raggiunge mai la seguente configurazione:

$$\exists \mathtt{x}, \mathtt{y}, \mathtt{z} \left(\begin{array}{l} \mathtt{x} \neq \mathtt{y} \wedge \mathtt{y} \neq \mathtt{z} \wedge \mathtt{z} \neq \mathtt{x} \wedge \\ \mathtt{state}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{received}[\mathtt{x}] = \mathtt{true} \wedge \\ \mathtt{coord}[\mathtt{y}] = \mathtt{true} \wedge \mathtt{state}[\mathtt{z}] = \mathtt{false} \wedge \\ \mathtt{faulty}[\mathtt{z}] = \mathtt{false} \wedge \mathtt{estimate}[\mathtt{y}]! = \mathtt{estimate}[\mathtt{z}] \end{array} \right)$$

Questa configurazione unsafe è stata messa al posto di quella riportata in Sezione A.4 (ovviamente togliendo i *system axiom* relativi ai tre lemmi). Alcuni dati relativi alla verifica:

- Numero di variabili nella formalizzazione: 15;
- Numero di transizioni nella formalizzazione: 28;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 4;
- Tempo: 12,53 secondi
- Massima profondità: 15
- Numero di nodi generati: 95
- Nodi cancellati: 10
- Chiamate all'SMT-solver: 11.590
- Invarianti trovati: 19

Secondo lemma

Lemma 7.3.3. *Sia q un processo corretto ed indeciso che imposta nel round 2 dell'algoritmo le variabili* $(\texttt{estimate}_q, \texttt{coord} - \texttt{id}_q)$ *a qualche valore,* (v, r)*. Finché q non riceve un messaggio di decide, tutti i coordinatori inoltreranno v come stima.*

Per esprimere questo lemma è stata introdotta una nuova variabile "storica" booleana che assume il valore `true` se il processo in questione ha già ricevuto (almeno una) stima. La variabile in questione è `processReceivedEstimate`. Il Lemma 7.3.3 implica che nel sistema non sarà mai vera la formula

$$\exists \texttt{x}, \texttt{y} \left(\begin{array}{l} \texttt{x} \neq \texttt{y} \wedge \texttt{round}[\texttt{x}] > 1 \wedge \texttt{state}[\texttt{x}] = \texttt{false} \wedge \\ \texttt{faulty}[\texttt{x}] = \texttt{false} \wedge \texttt{processReceivedEstimate}[\texttt{x}] = \texttt{true} \wedge \\ \texttt{coord}[\texttt{y}] = \texttt{true} \wedge \texttt{estimate}[\texttt{x}]! = \texttt{estimate}[\texttt{y}] \end{array} \right)$$

Come per il precedente lemma, anche questo è stato verificato inserendo la nuova configurazione unsafe al posto di quella riportata in Sezione A.4 (sempre togliendo i *system axiom* relativi ai tre lemmi). Alcuni dati relativi alla verifica:

- Numero di variabili nella formalizzazione: 15;
- Numero di transizioni nella formalizzazione: 28;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 5;
- Tempo: 386,62 secondi
- Massima profondità: 24
- Numero di nodi generati: 1.813
- Nodi cancellati: 215
- Chiamate all'SMT-solver: 261.814
- Invarianti trovati: 19

Terzo lemma

Lemma 7.3.4. *Se il coordinatore c è corretto, tutti i processi corretti decidono entro la fine della fase di c.*

Questo Lemma indica che la seguente formula è insoddisfacibile in ogni stato raggiungibile dal sistema:

$$\exists \mathtt{x}, \mathtt{y} \left(\begin{array}{l} \mathtt{x} \neq \mathtt{y} \wedge \mathtt{coord}[\mathtt{x}] = \mathtt{true} \wedge \mathtt{faulty}[\mathtt{x}] = \mathtt{false} \wedge \\ \mathtt{round}[\mathtt{x}] = 6 \wedge \mathtt{state}[\mathtt{y}] = \mathtt{false} \wedge \mathtt{faulty}[\mathtt{y}] = \mathtt{false} \wedge \end{array} \right)$$

Anche in questo caso la formalizzazione utilizzata per la verifica del lemma è quella riportata in Sezione A.4 senza i tre *system axiom* relativi ai lemmi e questi sono i dati relativi alla verifica:

- Numero di variabili nella formalizzazione: 15;

- Numero di transizioni nella formalizzazione: 28;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 2;
- Tempo: 1,46 secondi
- Massima profondità: 3
- Numero di nodi generati: 4
- Nodi cancellati: 0
- Chiamate all'SMT-solver: 2.414
- Invarianti trovati: 19

7.3.3 Tempi ed altri dati relativi alla verifica

Aggiungendo i tre *system axiom* si ha un ulteriore salvaguardia in termini di spazio: la verifica del protocollo formalizzato secondo le specifiche riportate in Sezione A.4 ha richiesto 4.719,51 secondi.[6] Altri dati relativi alla verifica:

- Numero di variabili nella formalizzazione: 15;
- Numero di transizioni nella formalizzazione: 28;
- Massimo numero di processi utilizzati per rappresentare le configurazioni: 6;
- Massima profondità: 39
- Numero di nodi generati: 11.158
- Nodi cancellati: 1.290

[6]Se si confronta questo dato con il precedente si nota un leggero peggioramento. Tuttavia si tratta di un peggioramento minimo, quasi 56 secondi su 1 ora e 15 minuti circa di tempo richiesto per la verifica, quindi questo ritardo potrebbe dipendere tranquillamente dalla gestione dello scheduling del sistema operativo, oppure da altre questioni che sono completamente indipendenti dal tool!

- Chiamate all'SMT-solver: 2.558.986
- Invarianti trovati: 19

In Appendice B sono riportati tutti i risultati di tutte le verifiche effettuate.

Algoritmo 7.1.1: Algoritmo per il Reliable Broadcast in caso di fallimenti *send-omission*.

Inizializzazione

$(\texttt{estimate}_p, \texttt{coord} - \texttt{id}_p) \leftarrow \begin{cases} (\texttt{m}, 0) & \text{se } p \text{ è il } \textit{sender} \ (m\text{: valore inizializzato dal sender}), \\ (\perp, -1) & \text{altrimenti.} \end{cases}$

$\texttt{state}_p \leftarrow \texttt{undecided}$;

for $c \leftarrow 1, 2, \ldots, t+1$ **do**

`/* Il processo` c `diventa coordinatore per tre round */`

Round 1

Tutti i processi p *indecisi* inviano $(\texttt{estimate}_p, \texttt{coord} - \texttt{id}_p)$ a c

if c *non riceve alcun messaggio di request* **then**

salta i round 2, 3 e 4

else

$\texttt{estimate}_c \leftarrow \texttt{estimate}_p$ alla quale è associato il più grande $\texttt{coord\_id}_p$

Round 2

c invia in modalità broadcast $(\texttt{estimate}_c, c)$

Tutti i processi p *indecisi* che hanno ricevuto $(\texttt{estimate}_c, c)$ impostano $(\texttt{estimate}_p, \texttt{coord} - \texttt{id}_p) \leftarrow (\texttt{estimate}_c, c)$

Round 3

Tutti i processi p *indecisi* che non hanno ricevuto $\texttt{estimate}_c$ nel Round 2 inviano un `nack` a c

Round 4

if c *non ha ricevuto alcun messaggio* ***nack*** **then**

invia in modalità broadcast il messaggio *decide*

else

$q_{crash}(c) = 1$ `/* Il coordinatore rileva il suo fallimento */`

Tutti i processi p *indecisi* che hanno ricevuto il messaggio *decide* eseguono:

$\texttt{decision}_p \leftarrow \texttt{estimate}_p$

$\texttt{state}_p \leftarrow \texttt{decided}_p$

8

Conclusioni e sviluppi futuri

Nel presente lavoro di tesi è stato presentato un linguaggio formale che permette di esprimere in maniera astratta e precisa un'ampia classe di sistemi fault-tolerant della quale fanno parte i protocolli studiati in dettagli nei capitoli 5, 6 e 7. La flessibilità di questo linguaggio permette la formalizzazione non solo di protocolli di rete, ma anche di sistemi concorrenti, funzioni scritte secondo un paradigma di programmazione imperativo e molti altri sistemi informatici eterogenei fra loro. La semplicità d'utilizzo e la flessibilità di un linguaggio formale sono due caratteristiche molto importanti: la modellizzazione di un sistema comporta di per sé un certo grado di astrazione e di arbitrarietà, in quanto solo le caratteristiche che si ritengono rilevanti del sistema vengono inserite nel modello. Astraendo dalle specifiche del sistema tuttavia è facile introdurre errori semantici che si riflettono nel processo di verifica falsandone i risultati: quanto più il sistema da formalizzare è complesso, tanto più il processo di formalizzazione è difficile e lungo, e ciò implica una probabilità di errore sempre più alta! Questa difficoltà è stata riscontrata durante l'attività sperimentale del lavoro di tesi e sarebbe auspicabile, in futuro, poter formalizzare i sistemi in maniera del tutto automatica, eventualmente implementando appositi compilatori per linguaggi largamente utilizzati al giorno d'oggi: una traduzione automatica (ovviamente accompagnata da una fase di preprocessamento della formalizzazione per astrarre da dettagli implementativi inutili a livello di verifica formale) eviterebbe di introdurre errori nella formalizzazione, garantendone la correttezza.

Per quanto riguarda la parte sperimentale, MCMT è stato utilizzato per verificare formalmente protocolli mai verificati prima. Per i primi tre protocolli studiati la verifica è stata eseguita in modalità del tutto automatica, e questo è un punto molto importante: l'interazione con l'utente può essere causa di errori o inesattezze che condurrebbero a risultati finali sbagliati. Per l'ultimo protocollo studiato, invece, è stata richiesta un'interazione con l'utente per indirizzare correttamente il processo di verifica. Tuttavia, grazie all'approccio completamente dichiarativo adottato da MCMT, l'interazione si è rivelata una semplice ricerca di candidati invarianti, che sono stati comunque verificati da MCMT. In questo senso, la correttezza del risultato finale non può essere messa in discussione in quanto l'utente non fa altro che *indirizzare* la verifica, e quindi non può introdurre errori che si riflettono nel risultato finale: un errore in questo senso implicherebbe solo performance peggiori, ma non risultati sbagliati! Questo ultimo punto, mette in luce un grande punto di forza di MCMT, ma allo stesso tempo ne evidenzia una carenza: la ricerca degli invarianti talvolta si rivela poco efficiente e deve essere affiancata da una ricerca *manuale* degli stessi. Un primo possibile sviluppo futuro potrebbe mirare allo studio di euristiche migliori per la sintesi degli invarianti; inoltre ad oggi (versione 1.0.1) MCMT è in grado di sintetizzare solo invarianti a due variabili e sarebbe auspicabile estendere il processo di sintesi ad invarianti con un numero qualsiasi di variabili (ovviamente, parallelamente alla sintesi di invarianti a più variabili bisognerà estendere la procedura di istanziazione degli stessi).

In futuro sarà necessaria una routine per l'analisi delle tracce spurie: la versione 1.0.1 del tool adotta il modello *stopping failures* per gestire transizioni con guardie universali. Questo modello è realistico e la sua adozione è giustificabile se MCMT viene utilizzato per verificare sistemi distribuiti, codice di programmi concorrenti ed altri esempi simili, ma non è assolutamente accettabile quando si verificano pezzi di codice scritto secondo il paradigma di programmazione imperativo, come per esempio funzioni di ricerca all'interno di liste, ordinamento di elementi in un array, ecc... In questo caso il tool potrebbe dare un risultato di non correttezza (cioè `UNSAFE`) proprio a causa dell'adozione del modello stopping failure. Una procedura per l'analisi di tracce spurie permetterebbe di rintracciare questi messaggi erronei e

continuare l'algoritmo di raggiungibilità a ritroso.

Sempre in relazione alle tracce unsafe, sarebbe molto utile realizzare un analizzatore di queste ultime in grado di derivare una rappresentazione "espansa", dove si osserva il contenuto di ogni variabile prima e dopo l'applicazione di tutte le transizioni della traccia in analisi (come quella riportata in Appendice C). Questo tool avrebbe una duplice applicazione: da un lato assumerebbe la funzione di "debugger", specialmente se le variabili della formalizzazione hanno lo stesso nome delle variabili del codice del sistema da verificare, e dall'altro potrebbe essere utilizzato per risolvere eventuali problemi di planning. In questi ultimi, infatti, i tool di model checking vengono utilizzati per mostrare come arrivare ad una configurazione unsafe rappresentante una particolare assegnazione delle variabili del sistema che risolve un dato problema di planning.

Un altro aspetto fondamentale nello sviluppo futuro di MCMT consiste nell'importazione, nell'adattamento e nel potenziamento di tecniche tradizionali utilizzate nel model checking per sistemi sequenziali, come ad esempio *widening*, *acceleration*, *predicate abstraction* e *ciclo CEGAR*. Alcune di queste tecniche sono già implementate nella versione 1.0 di *mcmt*, ma in modo ancora molto parziale.

In definitiva, l'approccio completamente dichiarativo alla base di MCMT è estremamente vantaggioso per descrivere formalmente sistemi informatici. I risultati sperimentali ottenuti in questa tesi fanno ben sperare sull'applicabilità di MCMT in moltissimi ambiti e sicuramente future attività di sviluppo di moduli aggiuntivi e l'adozione di strutture dati più performanti di quelle attuali permetteranno di ottenere risultati sempre più soddisfacenti.

A

Codice per MCMT

Riporto la formalizzazione dei quattro protocolli analizzati nella tesi. Questo codice può essere tradotto utilizzando il traduttore descritto nel Capitolo 3 per ottenere i file di specifica scritti secondo la sintassi accettata da MCMT.

A.1 Protocollo 1

Protocollo descritto in Capitolo 5.

```
/* Round of the execution */
global nat round [nat]
/* estimate of the processes (true = m; false = undefined) */
local bool estimate [nat]
/* state of the processes (true = decided; false = undecided) */
local bool state [nat]
/* who's the coordinator */
local bool coord [nat]
/* who has already been coordinator */
local bool aCoord [nat]
/* processes that has done the operations of the round */
local bool done [nat]
/* someone sent a request? */
global bool request [nat]
/* decision value of the processes (as estimate)*/
local bool decisionValue [nat]

/* initial configuration */
initial (universal x:nat) {
  (round[x] = 1) AND (state[x] = false) AND (coord[x] = false) AND
  (aCoord[x] = false) AND (done[x] = false) AND (request[x] = false)
}

/* unsafe configuration (uniform agreement) */
```

```
unsafe (existential x:nat, existential y:nat) {
  (state[x] = true) AND (state[y] = true) AND (decisionValue[x] != decisionValue[y])
}

/* 1) An undecided process sends a request message to the coordinator */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    request := true;
    done[x] := true;
}
/* 2) If the coordinator is not decided sends a request to himself */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (coord[x] = true) AND (done[x] = false) AND (state[x] = false)
  update:
    request := true;
    done[x] := true;
}
/* 3) If the coordinator received at least one request the system goes in round 2 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = true) AND (coord[x] = true) AND (done[x] = true)
  uguard: (done[all] = true)
  update:
    round := 2;
    done := lambda (j:nat) { false }
}
/* 4) If the coordinator didn't received any request is dismissed */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = false) AND (coord[x] = true) AND (done[x] = true)
  uguard: (done[all] = true)
  update:
    coord[x] := false;
    aCoord[x] := true;
}
/* 5) The coordinator sends his estimate to an undecided process */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    estimate[x] := estimate[y];
    done[x] := true;
}
/* 6) The undecided coordinator does nothing in this round */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (state[x] = false) AND (done[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
}
/* 7) Round 2 completed. System goes to round 3 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 2) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 3;
    done := lambda (j:nat) { false }
}
/* 8) Coordinator sends a 'decide' message to an undecided process */
transition (existential x:nat, existential y:nat) {
```

```
  guard: (round[x] = 3) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* 9) If the coordinator is undecided takes a decision. */
transition (existential x:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* 10) Round 3 completed. System goes to round 4 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 3) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 4;
    done := lambda (j:nat) { false }
}
/* 11) The coordinator in office is dismissed, a new process is elected as coordinator.
       The system restarts from round 1 */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 4) AND (coord[x] = true) AND (coord[y] = false) AND (aCoord[y] = false)
  update:
    round := 1;
    coord[x] := false;
    aCoord[x] := true;
    done := lambda (j:nat) { false }
    request := false;
    lambda (j:nat; x != j) {
      case y = j:
        coord[j] := true;
      otherwise:
        coord[j] := coord[j];
    }
}
/* 12) If there's no coordinator (maybe the coordinator in office crashed)
       a new one is elected and the system restarts from round 1 */
transition (existential x:nat, universal all:nat) {
  guard: (coord[x] = false) AND (aCoord[x] = false)
  uguard: (coord[all] = false)
  update:
    round := 1;
    coord[x] := true;
    done := lambda (j:nat) { false }
    request := false;
}
/* 13) Decided processes follow undecided ones doing nothing */
transition (existential x:nat, existential y:nat) {
  guard: (state[x] = true) AND (done[x] = false)
  update:
    done[x] := true;
}
```

A.2 Protocollo 2

Primo protocollo descritto in Capitolo 6.

```
/* Round of the execution */
global nat round [nat]
/* estimate of the processes (true = m; false = undefined) */
local bool estimate [nat]
/* state of the processes (true = decided; false = undecided) */
local bool state [nat]
/* who's the coordinator */
local bool coord [nat]
/* who has already been coordinator */
local bool aCoord [nat]
/* processes that has done the operations of the round */
local bool done [nat]
/* someone sent a request? */
global bool request [nat]
/* decision value of the processes (as estimate)*/
local bool decisionValue [nat]
/* the process is faulty (as defined in send omission failure model) */
local bool faulty [nat]

/* initial configuration */
initial (universal x:nat) {
  (round[x] = 1) AND (state[x] = false) AND (coord[x] = false) AND
  (aCoord[x] = false) AND (done[x] = false) AND (request[x] = false) AND
  (faulty[x] = false)
}

/* unsafe configuration (agreement) */
unsafe (existential x:nat, existential y:nat) {
  (state[x] = true) AND (faulty[x] = false) AND (state[y] = true) AND
  (faulty[y] = false) AND (decisionValue[x] != decisionValue[y])
}

/* 1) An undecided process sends a request message to the coordinator */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    request := lambda (j:nat) { true }
    done[x] := true;
}
/* 2) A faulty undecided process fails sending the request message */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    faulty[x] := true;
    done[x] := true;
}
/* 3) An undecided coordinator sends a request to himself */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    request := lambda (j:nat) { true }
    done[x] := true;
```

```
}
/* 4) If the coordinator received at least one request
      the system goes in round 2 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = true) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 2;
    done := lambda (j:nat) { false }
}
/* 5) If the coordinator didn't received any request is dismissed */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = false) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    coord[x] := false;
    aCoord[x] := true;
}
/* 6) The coordinator sends his estimate to an undecided process */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    estimate[x] := estimate[y];
    done[x] := true;
}
/* 7) A faulty coordinator fails sending his estimate */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; x != j) {
      case y = j :
        faulty[j] := true;
      otherwise :
        faulty[j] := faulty[j];
    }
}
/* 8) The coordinator does nothing in this round */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
}
/* 9) Round 2 completed. System goes to round 3 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 2) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 3;
    done := lambda (j:nat) { false }
}
/* 10) Coordinator sends a 'decide' message to an undecided process */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    state[x] := true;
    done[x] := true;
```

```
    decisionValue[x] := estimate[x];
}
/* 11) Coordinator fails sending a 'decide' */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; x != j) {
      case (j = y): faulty[j] := true;
      otherwise: faulty[j] := faulty[j];
    }
}
/* 12) An undecided coordinator takes a decision. */
transition (existential x:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* 13) Round 3 completed. System goes to round 4 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 3) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 4;
    done := lambda (j:nat) { false }
}
/* 14) The coordinator in office is dismissed, a new process is elected as coordinator.
       The system restarts from round 1 */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 4) AND (coord[x] = true) AND (coord[y] = false) AND (aCoord[y] = false)
  update:
    round := 1;
    coord[x] := false;
    aCoord[x] := true;
    done := lambda (j:nat) { false }
    request := false;
    lambda (j:nat; x != j) {
      case y = j:
        coord[j] := true;
      otherwise:
        coord[j] := coord[j];
    }
}
/* 15) If there's no coordinator (maybe the coordinator in office crashed)
       a new one is elected and the system restarts from round 1 */
transition (existential x:nat, universal all:nat) {
  guard: (coord[x] = false) AND (aCoord[x] = false)
  uguard: (coord[all] = false)
  update:
    round := 1;
    coord[x] := true;
    done := lambda (j:nat) { false }
    request := false;
}
/* 16) Decided processes follow undecided ones doing nothing */
```

```
transition (existential x:nat) {
  guard: (state[x] = true) AND (done[x] = false)
  update:
    done[x] := true;
}
```

A.3 Protocollo 3

Secondo protocollo descritto in Capitolo 6.

```
/* Round of the execution */
global nat round [nat]
/* estimate of the processes (true = m; false = undefined) */
local bool estimate [nat]
/* state of the processes (true = decided; false = undecided) */
local bool state [nat]
/* who's the coordinator */
local bool coord [nat]
/* who has already been coordinator */
local bool aCoord [nat]
/* processes that has done the operations of the round */
local bool done [nat]
/* someone sent a request? */
global bool request [nat]
/* decision value of the processes (as estimate)*/
local bool decisionValue [nat]
/* the process is faulty (as defined in send omission failure model) */
local bool faulty [nat]
/* the process has received the estimate from the coordinator */
local bool received [nat]
/* if nack[x] = true, process x received a negative ack */
local bool nack [nat]

/* initial configuration */
initial (universal x:nat) {
  (round[x] = 1) AND (state[x] = false) AND (coord[x] = false) AND
  (aCoord[x] = false) AND (done[x] = false) AND (request[x] = false) AND
  (faulty[x] = false) AND (received[x] = false) AND (nack[x] = false)
}

/* unsafe configuration (agreement) */
unsafe (existential x:nat, existential y:nat) {
  (state[x] = true) AND (faulty[x] = false) AND
  (state[y] = true) AND (faulty[y] = false) AND
  (decisionValue[x] != decisionValue[y])
}

/* 1) An undecided process sends a request message to the coordinator */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    request := true;
    done[x] := true;
}
```

```
/* 2) A faulty undecided process fails sending the request message */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    faulty[x] := true;
    done[x] := true;
}
/* 3) An undecided coordinator sends a request to himself */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    request := true;
    done[x] := true;
}
/* 4) If the coordinator received at least one request
      all the processes go in round 2 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = true) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 2;
    done := lambda (j:nat) { false }
}
/* 5) If the coordinator didn't received any request is dismissed */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (request[x] = false) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    coord[x] := false;
    aCoord[x] := true;
}
/* 6) The coordinator sends his estimate to an undecided process */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    estimate[x] := estimate[y];
    done[x] := true;
    received[x] := true;
}
/* 7) A faulty coordinator fails sending his estimate */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; x != j) {
      case y = j :
        faulty[j] := true;
      otherwise :
        faulty[j] := faulty[j];
    }
}
/* 8) The coordinator does nothing in this round */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
}
```

```
/* 9) Round 2 completed. System goes to round 3 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 2) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 3;
    done := lambda (j:nat) { false }
}
/* 10) If an undecided process didn't received the estimate sends a nack to the coordinator */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND (done[x] = false) AND
         (received[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; j != x) {
      case (j = y):
        nack[j] := true;
      otherwise:
        nack[j] := nack[j];
    }
}
/* 11) A faulty process who didn't received the estimate fails sending a nack */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND (done[x] = false) AND
         (received[x] = false) AND (coord[y] = true)
  update:
    faulty[x] := true;
    done[x] := true;
}
/* 12) If an undecided process received the estimate does nothing */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND (done[x] = false) AND
         (received[x] = true) AND (coord[y] = true)
  update:
    done[x] := true;
}
/* 13) The coordinator does nothing in this round */
transition (existential x:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND (done[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
}
/* 14) Round 3 completed, system goes to round 4 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 3) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 4;
}
/* 15) If the coordinator didn't received any nack the system goes to the 5th round. */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 4) AND (coord[x] = false)
  uguard: (nack[all] = false)
  update:
    round := 5;
    done := lambda (j:nat) { false }
}
```

```
/* 16) The coordinator sends a decide message */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 5) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* 17) The coordinator fails sending the decide message */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 5) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; x != j) {
      case (j = y):
        faulty[j] := true;
      otherwise:
        faulty[j] := faulty[j];
    }
}
/* 18) If the coordinator is undecided takes a decision. */
transition (existential x:nat) {
  guard: (round[x] = 5) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* 19) Round 5 completed, system goes to round 6 */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 5) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 6;
    done := lambda (j:nat) { false }
}
/* 20) The coordinator in office is dismissed, a new process
       is elected as coordinator. The system restarts from round 1 */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 6) AND (coord[x] = true) AND (coord[y] = false) AND (aCoord[y] = false)
  update:
    round := 1;
    coord[x] := false;
    aCoord[x] := true;
    received := lambda (j:nat) { false }
    request := false;
    done := lambda (j:nat) { false }
    nack := lambda (j:nat) { false }
    lambda (j:nat; x != j) {
      case y = j:
        coord[j] := true;
      otherwise:
        coord[j] := coord[j];
    }
}
/* 21) If there's no coordinator (maybe the coordinator in office crashed)
       a new one is elected and the system restarts from round 1 */
```

```
transition (existential x:nat, universal all:nat) {
  guard: (coord[x] = false) AND (aCoord[x] = false)
  uguard: (coord[all] = false)
  update:
    round := 1;
    coord[x] := true;
    done := lambda (j:nat) { false }
    request := false;
    received := lambda (j:nat) { false }
    nack := lambda (j:nat) { false }
}
/* 22) Decided processes follow undecided ones doing nothing */
transition (existential x:nat) {
  guard: (state[x] = true) AND (done[x] = false)
  update:
    done[x] := true;
}
```

A.4 Protocollo 4

Protocollo descritto in Capitolo 7. Il codice riportato in questa Sezione permette di ottenere le statistiche riportate in Sezione7.3.

```
/* Round of the execution */
global nat round [nat]
/* estimate of the processes (true = m; false = undefined) */
local bool estimate [nat]
/* state of the processes (true = decided; false = undecided) */
local bool state [nat]
/* who's the coordinator */
local bool coord [nat]
/* who has already been coordinator */
local bool aCoord [nat]
/* processes that has done the operations of the round */
local bool done [nat]
/* someone sent a request? */
global bool request [nat]
/* decision value of the processes (as estimate)*/
local bool decisionValue [nat]
/* the process is faulty (as defined in send omission failure model) */
local bool faulty [nat]
/* the process has received the estimate from the coordinator */
local bool received [nat]
/* if nack[x] = true, process x received a negative ack */
local bool nack [nat]
/* id of the coordinator who sent me an estimate */
local int id [nat]
/* id of the coordinator who sent the estimate saved in tmpEstimate */
global int maxId [nat]
/* temp. var used to send the estimate to the coordinator */
global bool tmpEstimate [nat]
/* used for the Lemma 2.2 of the paper */
local bool processReceivedEstimate [nat]
```

```
/* optimization (available from 1.0.1 version of mcmt) */
key_search round;

/* tells to mcmt to use at most 4 vars of sort INDEX in invariants verification */
(:mcmt :inv_search_max_index_var 4 :)

/* constraints for numerical vars */
system_axiom (universal x:nat) {
  (round[x] < 8) AND ( (maxId[x] + 3) > 0) AND ( (id[x] + 2) > 0)
}

/* Exists only one coordinator in the system */
suggested_negated_invariant (existential x:nat, existential y:nat) {
  (coord[x] = true) AND (coord[y] = true)
}
/* If 'c' is coordinator, all other process with id < c have already been coordinator */
suggested_negated_invariant (existential x:nat, existential y:nat) {
  (coord[x] = true) AND (aCoord[y] = false) AND (y < x)
}
/* A coordinator can't have id[] greater than his identificator */
suggested_negated_invariant (existential x:nat) {
  (coord[x] = true) AND (id[x] > x)
}
/* A process can't have id[] greater than coordinator's identificator */
suggested_negated_invariant (existential x:nat, existential y:nat) {
  (coord[x] = true) AND (id[y] > x)
}
/* In the first round a process can't have id[] equals to the coordinator's identificator */
suggested_negated_invariant (existential x:nat, existential y:nat) {
  (round[x] = 1) AND (coord[x] = true) AND (id[y] = x)
}
/* A correct process can't receive any nack */
suggested_negated_invariant (existential x:nat) {
  (faulty[x] = false) AND (nack[x] = true)
}
/* Coordinators are elected in order by identificator */
suggested_negated_invariant (existential x:nat, existential y:nat) {
  (coord[x] = true) AND (aCoord[y] = true) AND (y > x)
}

/* LEMMA 2.1 */
system_axiom (universal x:nat, universal y:nat, universal z:nat) {
  (NOT ( (state[x] = true) AND (received[x] = true) AND (coord[y] = true) AND
  (state[z] = false) AND (faulty[z]= false) AND (estimate[y] != estimate[z]) AND
  (x != y) AND (y != z) AND (z != x) ))
}
/* LEMMA 2.2 */
system_axiom (universal x:nat, universal y:nat) {
  (NOT ( (round[x] > 1) AND (state[x] = false) AND (faulty[x] = false) AND
  (processReceivedEstimate[x] = true) AND (coord[y] = true) AND
  (estimate[x] != estimate[y]) AND (x != y) ))
}
/* LEMMA 2.3 */
system_axiom (universal x:nat, universal y:nat) {
  (NOT ( (coord[x] = true) AND (faulty[x] = false) AND (round[x] = 6) AND
  (state[y] = false) AND (faulty[y] = false) AND (x != y) ))
```

```
}

/* initial configuration */
initial (universal x:nat) {
  (round[x] = 1) AND (state[x] = false) AND (coord[x] = false) AND
  (aCoord[x] = false) AND (done[x] = false) AND (received[x] = false) AND
  (nack[x] = false) AND (id[x] = -1) AND (maxId[x] = -2) AND
  (processReceivedEstimate[x] = false)
}

/* AGREEMENT - complete */
unsafe (existential x:nat, existential y:nat) {
  (state[x] = true) AND (faulty[x] = false) AND
  (state[y] = true) AND (faulty[y] = false) AND
  (decisionValue[x] != decisionValue[y])
}

/* 1) Decided processes send request to coordinator if their id are greather than maxId */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND
         (coord[y] = true) AND (id[x] > maxId[x])
  update:
    done[x] := true;
    maxId := id[x];
    tmpEstimate := estimate[x];
}
/* 2) Undecided processes want to send a request, but their id is less or equal than maxId */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND
         (coord[y] = true) AND (id[x] <= maxId[x])
  update:
    done[x] := true;
}
/* 3) An undecided faulty process fails sending a request */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND
         (coord[y] = true) AND (faulty[x] = true)
  update:
    done[x] := true;
}
/* 4) An undecided coordinator (with id > maxId) sends a request (to himself) */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND
         (coord[x] = true) AND (id[x] > maxId[x])
  update:
    done[x] := true;
    maxId := id[x];
    tmpEstimate := estimate[x];
}
/* 5) An undecided coordinator (with id <= maxId) doesn't send a request to himself */
transition (existential x:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = false) AND
         (coord[x] = true) AND (id[x] <= maxId[x])
  update:
    done[x] := true;
}
/* 6) Decided processes do nothing */
```

```
transition (existential x:nat) {
  guard: (round[x] = 1) AND (done[x] = false) AND (state[x] = true)
  update:
    done[x] := true;
}
/* 7) If all processes have completed the 1st round and the coordinator
      received a request (i.e. maxId > -2), the coordinator set his estimate
      to tmpEstimate value and all the processes go to 2nd round */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (maxId[x] > -2) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 2;
    done := lambda (j:nat) { false }
    estimate[x] := tmpEstimate[x];
}
/* 8) ... otherwise (no request have been received by the coordinator)
      the coordinator is dismissed. */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 1) AND (maxId[x] = -2) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    coord[x] := false;
    aCoord[x] := true;
}
/* 9) An undecided process receives the estimate from the coordinator. */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    estimate[x] := estimate[y];
    done[x] := true;
    received[x] := true;
    id[x] := y;
    processReceivedEstimate[x] := true;
}
/* 10) A faulty coordinator fails sending an estimate */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND
         (coord[y] = true) AND (faulty[y] = true)
  update:
    done[x] := true;
}
/* 11) If the coordinator is undecided sends an estimate to himself */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
    received[x] := true;
    id[x] := x;
    processReceivedEstimate[x] := true;
}
/* 12) Decided processes do nothing */
transition (existential x:nat) {
  guard: (round[x] = 2) AND (done[x] = false) AND (state[x] = true)
  update:
    done[x] := true;
}
```

```
/* 13) Round 2 completed. System goes to round 3. */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 2) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 3;
    done := lambda (j:nat) { false }
}
/* 14) If an undecided process didn't received the estimate sends a nack to coord */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND (done[x] = false) AND
         (received[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
    lambda (j:nat; j != x) {
      case (j = y):
        nack[j] := true;
      otherwise:
        nack[j] := nack[j];
    }
}
/* 15) An undecided faulty process that didn't received the estimate
       fails sending the nack to the coordinator */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND (done[x] = false) AND
         (received[x] = false) AND (coord[y] = true) AND (faulty[x] = true)
  update:
    done[x] := true;
}
/* 16) Af an undecided process has received the estimate does nothing */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = false) AND (done[x] = false) AND
         (received[x] = true) AND (coord[y] = true)
  update:
    done[x] := true;
}
/* 17) the coordinator does nothing in this round */
transition (existential x:nat) {
  guard: (round[x] = 3) AND (done[x] = false) AND (coord[x] = true)
  update:
    done[x] := true;
}
/* 18) decided processes do nothing in this round */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 3) AND (state[x] = true) AND (done[x] = false) AND (coord[y] = true)
  update:
    done[x] := true;
}
/* 19) When all the processes have done, system goes to the 4th round. */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 3) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 4;
}
/* 20) If no nacks have been received, system goes to the 5th round (if the coordinator
       received one (or more) nack, he is killed by the system) */
```

```
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 4) AND (coord[x] = false)
  uguard: (nack[all] = false)
  update:
    round := 5;
    done := lambda (j:nat) { false }
}
/* 21) Coordinator sends a decide to an undecided process */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 5) AND (done[x] = false) AND (state[x] = false) AND (coord[y] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* 22) A faulty coordinator fails sending the message */
transition (existential x:nat, existential y:nat) {
  guard: (round[x] = 5) AND (done[x] = false) AND (state[x] = false) AND
         (coord[y] = true) AND (faulty[y] = true)
  update:
    done[x] := true;
}
/* 23) A faulty coordinator sends a decide to himself */
transition (existential x:nat) {
  guard: (round[x] = 5) AND (done[x] = false) AND (state[x] = false) AND (coord[x] = true)
  update:
    state[x] := true;
    done[x] := true;
    decisionValue[x] := estimate[x];
}
/* 24) Coordinator sends a decide to decided processes
       (but they ignore this message) */
transition (existential x:nat) {
  guard: (round[x] = 5) AND (done[x] = false) AND (state[x] = true)
  update:
    done[x] := true;
}
/* 25) Round n. 5 completed. System goes to round 6. */
transition (existential x:nat, universal all:nat) {
  guard: (round[x] = 5) AND (done[x] = true) AND (coord[x] = true)
  uguard: (done[all] = true)
  update:
    round := 6;
    done := lambda (j:nat) { false }
}
/* 26) Coordinator in office is dismissed. */
transition (existential x:nat) {
  guard: (round[x] = 6) AND (coord[x] = true)
  update:
    round := 7;
    coord[x] := false;
    aCoord[x] := true;
}
/* 27) There's no coordinator! (maybe the coordinator crashed) */
transition (existential x:nat, universal y:nat) {
  guard: (coord[x] = false)
  uguard: (coord[y] = false)
```

```
  update:
    round := 7;
}
/* 28) In this round a new process is elected as coordinator of the system.
       Then the system goes to 1st round. */
transition (existential x:nat, universal y:nat) {
  guard: (coord[x] = false) AND (aCoord[x] = false) AND (round[x] = 7)
  uguard: (y >= x)
  uguard: (y < x) AND (aCoord[y] = true)
  update:
    round := 1;
    coord[x] := true;
    received := lambda (j:nat) { false }
    done := lambda (j:nat) { false }
    nack := lambda (j:nat) { false }
    maxId := -2;
}
```

B

Risultati sperimentali

In Tabella B.1 sono riportati tutti i dati speriementali ottenuti nelle varie verifiche dei protocolli. Per problemi di spazio, i protocolli sono rappresentati da un numero (prima colonna della tabella):

1. Prima verifica, protocollo descritto nel Capitolo 5, formalizzazione in Sezione A.1.

2. Seconda verifica, protocollo descritto nella Sezione 6.1 e 6.2, formalizzazione in Sezione A.2.

3. Terza verifica, protocollo descritto nella Sezione 6.3, formalizzazione in Sezione A.3.

4. Quarta verifica, protocollo descritto nel Capitolo 7, formalizzazione in Sezione A.4 (senza *system axiom* relativi ai lemmi).

5. Ultima verifica, protocollo descritto nel Capitolo 7, formalizzazione in Sezione A.4 (con i tre *system axiom* relativi ai lemmi).

6. Verifica del primo lemma, descritto in Sezione 7.3.2, formalizzazione in Sezione A.4 (sostituire la formula unsafe con quella del lemma in questione).

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
1	SAFE	4	8	13	1,18	13	113	21	2.792	×
2	UNSAFE	5	9	16	17,66	12	464	26	19733	×
3	UNSAFE	6	11	22	1.624,24	34	9.679	770	1.338.058	×
4	SAFE	6	15	28	4.663,74	39	11.206	1.290	2.563.756	19 (+7)
5	SAFE	6	15	28	4.719,51	39	11.158	1.290	2.558.986	19 (+7)
6	SAFE	4	15	28	12,53	15	95	10	11.590	19 (+7)
7	SAFE	5	15	28	386,62	24	1.813	215	261.814	19 (+7)
8	SAFE	2	15	28	1,46	3	4	0	2.414	19 (+7)

Tabella B.1: Tabella riassuntiva con tutti i dati realivi alle verifiche sperimentali.

7. Verifica del primo lemma, descritto in Sezione 7.3.2, formalizzazione in Sezione A.4 (sostituire la formula unsafe con quella del lemma in questione).

8. Verifica del primo lemma, descritto in Sezione 7.3.2, formalizzazione in Sezione A.4 (sostituire la formula unsafe con quella del lemma in questione).

Le colonne invece rappresentano questi valori:

(a) risultato della verifica

(b) massimo numero di processi utilizzati per rappresentare le configurazioni

(c) numero di variabili nella formalizzazione

(d) numero di transizioni nella formalizzazione

(e) tempo richiesto dalla verifica in secondi

(f) massima profondità raggiunta

(g) numero di nodi generati

(h) numero di nodi cancellati

(i) numero di chiamate all'SMT-solver

(j) invarianti trovati (× indica che non è stata abilitata la ricerca degli invarianti)

Questi dati sono stati ottenuti su una macchina con le seguenti caratteristiche:

- Processore: Intel(R) Core(TM)2 Duo CPU E7300 @ 2.66GHz
- Memoria: 2 GB
- Sistema operativo: Debian

C

Analisi delle "tracce unsafe"

Questa appendice è dedicata all'analisi della sequenza di transizioni trovata da MCMT durante la verifica dei protocolli riportati in Algoritmo 5.2.1 (assumento fallimenti in *send-omission*) e Algoritmo 6.3.1

Queste sequenze di transizioni rappresentano l'esecuzione che porta il sistema in uno stato finale, dove la proprietà di *agreement* non è soddisfatta.

Notazione Ogni configurazione che verrà mostrata è accompagnata da un'etichetta del tipo "$\tau_7(\texttt{p1}, \texttt{p3})$" per indicare che essa è ottenuta (dalla precedente) applicando la transizione 7 interpretando la prima variabile esistenzialmente quantificata come il processo `p1` e la seconda come il processo `p3`.

C.1 Prima traccia

Questa prima sezione è dedicata all'analisi della "traccia unsafe" mostrata da MCMT alla fine della verifica del protocollo riportato in Algoritmo 5.2.1 assumendo che nel sistema possano esserci alcuni processi che falliscono in *send-omission*.

I processi coinvolti nella dimostrazione sono tre (indicati con `p1`, `p2` e `p3`). Per ogni transizione applicata viene mostrato come cambia lo stato del sistema.

	p1	p2	p3
init	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false request = false faulty = false decisionValue =⊥	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false request = false faulty = false decisionValue =⊥	round = 1 estimate = m state = false coord = false aCoord = false done = false request = false faulty = false decisionValue =⊥
τ_{15}(p3)	**round = 1** estimate =⊥ state = false coord = false aCoord = false **done = false** **request = false** faulty = false decisionValue =⊥	**round = 1** estimate =⊥ state = false coord = false aCoord = false **done = false** **request = false** faulty = false decisionValue =⊥	**round = 1** estimate = m state = false **coord = true** aCoord = false **done = false** **request = false** faulty = false decisionValue =⊥
τ_3(p3)	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false decisionValue =⊥	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false decisionValue =⊥	round = 1 estimate = m state = false coord = true aCoord = false **done = true** **request = true** faulty = false decisionValue =⊥
τ_1(p2, p3)	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false decisionValue =⊥	round = 1 estimate =⊥ state = false coord = false aCoord = false **done = true** **request = true** faulty = false decisionValue =⊥	round = 1 estimate = m state = false coord = true aCoord = false done = true **request = true** faulty = false decisionValue =⊥

$\tau_1(\mathtt{p1}, \mathtt{p3})$	round = 1 estimate = $\perp$ state = false coord = false aCoord = false **done = true** **request = true** faulty = false decisionValue = $\perp$	round = 1 estimate = $\perp$ state = false coord = false aCoord = false done = true **request = true** faulty = false decisionValue = $\perp$	round = 1 estimate = m state = false coord = true aCoord = false done = true **request = true** faulty = false decisionValue = $\perp$
$\tau_4(\mathtt{p3})$	**round = 2** estimate = $\perp$ state = false coord = false aCoord = false **done = false** request = true faulty = false decisionValue = $\perp$	**round = 2** estimate = $\perp$ state = false coord = false aCoord = false **done = false** request = true faulty = false decisionValue = $\perp$	**round = 2** estimate = m state = false coord = true aCoord = false **done = false** request = true faulty = false decisionValue = $\perp$
$\tau_8(\mathtt{p3})$	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false decisionValue = $\perp$	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false decisionValue = $\perp$	round = 2 estimate = m state = false coord = true aCoord = false **done = true** request = true faulty = false decisionValue = $\perp$
$\tau_7(\mathtt{p2}, \mathtt{p3})$	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false decisionValue = $\perp$	round = 2 estimate = $\perp$ state = false coord = false aCoord = false **done = true** request = true faulty = false decisionValue = $\perp$	round = 2 estimate = m state = false coord = true aCoord = false done = true request = true **faulty = true** decisionValue = $\perp$

$\tau_6(\mathtt{p1}, \mathtt{p3})$	round = 2 **estimate = m** state = false coord = false aCoord = false **done = true** request = true faulty = false decisionValue = $\perp$	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = true request = true faulty = false decisionValue = $\perp$	round = 2 estimate = m state = false coord = true aCoord = false done = true request = true faulty = true decisionValue = $\perp$
$\tau_9(\mathtt{p3})$	**round = 3** estimate = m state = false coord = false aCoord = false **done = false** request = true faulty = false decisionValue = $\perp$	**round = 3** estimate = $\perp$ state = false coord = false aCoord = false **done = false** request = true faulty = false decisionValue = $\perp$	**round = 3** estimate = m state = false coord = true aCoord = false **done = false** request = true faulty = true decisionValue = $\perp$
$\tau_{10}(\mathtt{p2}, \mathtt{p3})$	round = 3 estimate = m state = false coord = false aCoord = false done = false request = true faulty = false decisionValue = $\perp$	round = 3 estimate = $\perp$ **state = true** coord = false aCoord = false **done = true** request = true faulty = false **decisionValue = $\perp$**	round = 3 estimate = m state = false coord = true aCoord = false done = false request = true faulty = true decisionValue = $\perp$
$\tau_{10}(\mathtt{p1}, \mathtt{p3})$	round = 3 estimate = m **state = true** coord = false aCoord = false **done = true** request = true **faulty = false** **decisionValue = m**	round = 3 estimate = $\perp$ **state = true** coord = false aCoord = false done = true request = true **faulty = false** **decisionValue = $\perp$**	round = 3 estimate = m state = false coord = true aCoord = false done = false request = true faulty = true decisionValue = $\perp$

C.2 Seconda traccia

Questa seconda sezione è dedicata all'analisi della "traccia unsafe" mostrata da MCMT alla fine della verifica del protocollo riportato in Algoritmo 6.3.1 assumendo che nel sistema possano esserci alcuni processi che falliscono in *send-omission*.

I processi coinvolti nella dimostrazione sono quattro (indicati con p1, p2, p3 e p4). Per ogni transizione applicata viene mostrato come cambia lo stato del sistema.

	p1	p2	p3	p4
init	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false request = false faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false request = false faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false request = false faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = false coord = false aCoord = false done = false request = false faulty = false received = false nack = false decidedValue =⊥ *crash* = false
τ_{21}(p4)	**round = 1** estimate =⊥ state = false coord = false aCoord = false **done = false** **request = false** faulty = false **received = false** **nack = false** decidedValue =⊥ *crash* = false	**round = 1** estimate =⊥ state = false coord = false aCoord = false **done = false** **request = false** faulty = false **received = false** **nack = false** decidedValue =⊥ *crash* = false	**round = 1** estimate =⊥ state = false coord = false aCoord = false **done = false** **request = false** faulty = false **received = false** **nack = false** decidedValue =⊥ *crash* = false	**round = 1** estimate = m state = false **coord = true** aCoord = false **done = false** **request = false** faulty = false **received = false** **nack = false** decidedValue =⊥ *crash* = false
τ_3(p4)	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = false coord = true aCoord = false **done = true** **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false

$\tau_1(\mathtt{p3},\mathtt{p4})$	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false **done = true** **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = false coord = true aCoord = false done = true **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false
$\tau_1(\mathtt{p2},\mathtt{p4})$	round = 1 estimate =⊥ state = false coord = false aCoord = false done = false **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false **done = true** **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = true **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = false coord = true aCoord = false done = truee **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false
$\tau_1(\mathtt{p1},\mathtt{p4})$	round = 1 estimate =⊥ state = false coord = false aCoord = false **done = true** **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = true **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate =⊥ state = false coord = false aCoord = false done = true **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = false coord = true aCoord = false done = true **request = true** faulty = false received = false nack = false decidedValue =⊥ *Crash* = false
$\tau_4(\mathtt{p4})$	**round = 2** estimate =⊥ state = false coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false	**round = 2** estimate =⊥ state = false coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false	**round = 2** estimate =⊥ state = false coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false	**round = 2** estimate = m state = false coord = true aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false

τ_8(p4)	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = false coord = true aCoord = false **done = true** request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false
τ_7(p3, p4)	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = $\perp$ state = false coord = false aCoord = false **done = true** request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = false coord = true aCoord = false done = true request = true **faulty = true** received = true nack = false decidedValue = $\perp$ *crash* = false
τ_6(p2, p4)	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 **estimate = m** state = false coord = false aCoord = false **done = true** request = true faulty = false **received = true** nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = true request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue = $\perp$ *crash* = false
τ_6(p1, p4)	round = 2 **estimate = m** state = false coord = false aCoord = false **done = true** request = true faulty = false **received = true** nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = false coord = false aCoord = false done = true request = true faulty = false received = true nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = $\perp$ state = false coord = false aCoord = false done = true request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue = $\perp$ *crash* = false

τ_9(p4)	**round = 3** estimate = m state = false coord = false aCoord = false **done = false** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 3** estimate = m state = false coord = false aCoord = false **done = false** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 3** estimate =⊥ state = false coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false	**round = 3** estimate = m state = false coord = true aCoord = false **done = false** request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false
τ_{13}(p4)	round = 3 estimate = m state = false coord = false aCoord = false done = false request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = false aCoord = false done = false request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate =⊥ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = true aCoord = false **done = true** request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false
τ_{12}(p2, p4)	round = 3 estimate = m state = false coord = false aCoord = false done = false request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = false aCoord = false **done = true** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate =⊥ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false
τ_{12}(p1, p4)	round = 3 estimate = m state = false coord = false aCoord = false **done = true** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = false aCoord = false done = true request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate =⊥ state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false

τ_{11}(p3, p4)	round = 3 estimate = m state = false coord = false aCoord = false done = true request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = false aCoord = false done = true request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate =⊥ state = false coord = false aCoord = false **done = true** request = true **faulty = true** received = false nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false
τ_{14}(p4)	**round = 4** estimate = m state = false coord = false aCoord = false done = true request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 4** estimate = m state = false coord = false aCoord = false done = true request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 4** estimate =⊥ state = false coord = false aCoord = false done = true request = true faulty = true received = false nack = false decidedValue =⊥ *crash* = false	**round = 4** estimate = m state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false
τ_{15}(p1)	**round = 5** estimate = m state = false coord = false aCoord = false **done = false** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 5** estimate = m state = false coord = false aCoord = false **done = false** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 5** estimate =⊥ state = false coord = false aCoord = false **done = false** request = true faulty = true received = false nack = false decidedValue =⊥ *crash* = false	**round = 5** estimate = m state = false coord = true aCoord = false **done = false** request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false
τ_{16}(p2, p4)	round = 5 estimate = m state = false coord = false aCoord = false done = false request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 5 estimate = m **state = true** coord = false aCoord = false **done = true** request = true faulty = false received = true nack = false **decidedValue = m** *crash* = false	round = 5 estimate =⊥ state = false coord = false aCoord = false done = false request = true faulty = true received = false nack = false decidedValue =⊥ *crash* = false	round = 5 estimate = m state = false coord = true aCoord = false done = false request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false

τ_{21}(p3)	**round = 1** estimate = m state = false coord = false aCoord = false **done = false** **request = false** faulty = false **received = false** **nack = false** decidedValue =⊥ *crash* = false	**round = 1** estimate = m state = true coord = false aCoord = false **done = false** **request = false** faulty = false **received = false** **nack = false** decidedValue = m *crash* = false	**round = 1** estimate =⊥ state = false **coord = true** aCoord = false **done = false** **request = false** faulty = true **received = false** **nack = false** decidedValue =⊥ *crash* = false	*Crash* = true
τ_{22}(p2)	round = 1 estimate = m state = false coord = false aCoord = false done = false request = false faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = true coord = false aCoord = false **done = true** request = false faulty = false received = false nack = false decidedValue = m *crash* = false	round = 1 estimate =⊥ state = false coord = true aCoord = false done = false request = false faulty = true received = false nack = false decidedValue =⊥ *crash* = false	*Crash* = true
τ_3(p3)	round = 1 estimate = m state = false coord = false aCoord = false done = false **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = true coord = false aCoord = false done = true **request = true** faulty = false received = false nack = false decidedValue = m *crash* = false	round = 1 estimate =⊥ state = false coord = true aCoord = false **done = true** **request = true** faulty = true received = false nack = false decidedValue =⊥ *crash* = false	*Crash* = true
τ_1(p1, p3)	round = 1 estimate = m state = false coord = false aCoord = false **done = true** **request = true** faulty = false received = false nack = false decidedValue =⊥ *crash* = false	round = 1 estimate = m state = true coord = false aCoord = false done = true **request = true** faulty = false received = false nack = false decidedValue = m *crash* = false	round = 1 estimate =⊥ state = false coord = true aCoord = false done = true **request = true** faulty = true received = false nack = false decidedValue =⊥ *crash* = false	*Crash* = true

τ_4(p3)	**round = 2** estimate = m state = false coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	**round = 2** estimate = m state = true coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue = m *crash* = false	**round = 2** estimate = $\perp$ state = false coord = true aCoord = false **done = false** request = true faulty = true received = false nack = false decidedValue = $\perp$ *crash* = false	*Crash* = true
τ_{22}(p2)	round = 2 estimate = m state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = true coord = false aCoord = false **done = true** request = true faulty = false received = false nack = false decidedValue = m *crash* = false	round = 2 estimate = $\perp$ state = false coord = true aCoord = false done = false request = false faulty = true received = false nack = false decidedValue = $\perp$ *crash* = false	*Crash* = true
τ_8(p3)	round = 2 estimate = m state = false coord = false aCoord = false done = false request = true faulty = false received = false nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = true coord = false aCoord = false done = true request = true faulty = false received = false nack = false decidedValue = m *crash* = false	round = 2 estimate = $\perp$ state = false coord = true aCoord = false **done = true** request = true faulty = true received = false nack = false decidedValue = $\perp$ *crash* = false	*Crash* = true
τ_6(p1, p3)	round = 2 **estimate = $\perp$** state = false coord = false aCoord = false **done = true** request = true faulty = false **received = true** nack = false decidedValue = $\perp$ *crash* = false	round = 2 estimate = m state = true coord = false aCoord = false done = true request = true faulty = false received = false nack = false decidedValue = m *crash* = false	round = 2 estimate = $\perp$ state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue = $\perp$ *crash* = false	*Crash* = true

τ_9(p3)	**round = 3** estimate =⊥ state = false coord = false aCoord = false **done = false** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 3** estimate = m state = true coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue = m *crash* = false	**round = 3** estimate =⊥ state = false coord = true aCoord = false **done = false** request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false	*Crash* = true
τ_{22}(p2)	round = 3 estimate =⊥ state = false coord = false aCoord = false done = false request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = true coord = false aCoord = false **done = true** request = true faulty = false received = false nack = false decidedValue = m *crash* = false	round = 3 estimate =⊥ state = false coord = true aCoord = false done = false request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false	*Crash* = true
τ_{13}(p3)	round = 3 estimate =⊥ state = false coord = false aCoord = false done = false request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = true coord = false aCoord = false done = true request = true faulty = false received = false nack = false decidedValue = m *crash* = false	round = 3 estimate =⊥ state = false coord = true aCoord = false **done = true** request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false	*Crash* = true
τ_{12}(p1, p3)	round = 3 estimate =⊥ state = false coord = false aCoord = false **done = true** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	round = 3 estimate = m state = true coord = false aCoord = false done = true request = true faulty = false received = false nack = false decidedValue = m *crash* = false	round = 3 estimate =⊥ state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false	*Crash* = true

τ_{14}(p3)	**round = 4** estimate =⊥ state = false coord = false aCoord = false done = true request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 4** estimate = m state = true coord = false aCoord = false done = true request = true faulty = false received = false nack = false decidedValue = m *crash* = false	**round = 4** estimate =⊥ state = false coord = true aCoord = false done = true request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false	$Crash$ = true
τ_{15}(p1)	**round = 5** estimate =⊥ state = false coord = false aCoord = false **done = false** request = true faulty = false received = true nack = false decidedValue =⊥ *crash* = false	**round = 5** estimate = m state = true coord = false aCoord = false **done = false** request = true faulty = false received = false nack = false decidedValue = m *crash* = false	**round = 5** estimate =⊥ state = false coord = true aCoord = false **done = false** request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false	$Crash$ = true
τ_{16}(p1, p3)	round = 5 estimate =⊥ **state = true** coord = false aCoord = false **done = true** request = true **faulty = false** received = true nack = false **decidedValue = ⊥** *crash* = false	round = 5 estimate = m **state = true** coord = false aCoord = false done = false request = true **faulty = false** received = false nack = false **decidedValue = m** *crash* = false	round = 5 estimate =⊥ state = false coord = true aCoord = false done = false request = true faulty = true received = true nack = false decidedValue =⊥ *crash* = false	$Crash$ = true

Bibliografia

[1] AAVV. Nusmv: a new symbolic model checker. `nusmv.irst.itc.it`.

[2] AAVV. Spin - formal verification. `spinroot.com`.

[3] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 313–321. IEEE Computer Society, 1996.

[4] P. A. Abdulla, G. Delzanno, N. H. Henda, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS*, pages 721–736, 2007.

[5] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, pages 145–157, 2007.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, & Tools*. Pearson Education, 2007.

[7] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[8] B. Bérard and L. Fribourg. Reachability analysis of (timed) petri nets using real arithmetic. In *CONCUR*, pages 178–193, 1999.

[9] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data. In *FCT*, pages 1–22, 2007.

[10] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transistion relations. In *VLSI*, pages 49–58, 1991.

[11] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[13] T. D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Proceedings of the 4th international workshop on Distributed algorithms*, pages 289–303, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[14] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1981.

[15] E. M. Clarke, Grumberg O., and Peled D. *Model Checking*. MIT Press, 1999.

[16] Herbert B. Enderton. *A mathematical Introduction to Logic*. Academic Press, New York-London, 1972.

[17] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[18] Free Software Foundation. GCC, the GNU Compiler Collection . `http://gcc.gnu.org/`, 2009.

[19] Python Software Foundation. Python Programming Language – Official Website. `http://www.python.org/`, 2009.

[20] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model-checking of array-based systems. In *Proc. of IJCAR*, LNCS, 2008.

[21] S. Ghilardi and S. Ranise. Goal-directed Invariant Synthesis for Model Checking Modulo Theories. In *Tableaux '09*. LNCS, 2009.

[22] S. Ghilardi and S. Ranise. MCMT: Model Checker Modulo Theories. `http://homes.dsi.unimi.it/~ghilardi/mcmt/`, 2009.

[23] S. Ghilardi and S. Ranise. Model Checking Modulo Theory at work: the integration of Yices in MCMT. In *AFM 09 (co-located with CAV 09)*, 2009.

[24] S. Ghilardi, S. Ranise, and T. Valsecchi. Light-weight smt-based model-checking. In *AVOCS 07-08*. ENTCS, 2008.

[25] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *CAV*, pages 72–83, 1997.

[26] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. *Distributed systems (2nd Ed.)*, pages 97–145, 1993.

[27] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Automi, linguaggi e calcolabilità*. Pearson Education Italia, 2003.

[28] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. `http://dinosaur.compilertools.net/yacc/index.html`.

[29] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[30] K. L. McMillan. Symbolic model checking: an approach to the state explosion problem, Kluwer Academic, 1993.

[31] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.

[32] The GNU Project. Bison, The YACC-compatible Parser Generator. http://www.gnu.org/software/bison/.

[33] Paxson Vern. flex: The Fast Lexical Analyzer. http://flex.sourceforge.net/.