UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
Dottorato di Ricerca in Ingegneria Informatica ed Automatica

UNA METODOLOGIA PER LA MODELLAZIONE FORMALE DI SISTEMI CRITICI BASATA SU METODI E TECNICHE DI MODEL DRIVEN ENGINEERING

CAMILLA PAPA

TESI DI DOTTORATO DI RICERCA
(XXIII CICLO)
NOVEMBRE 2011

Il Tutore

Prof.ssa Valeria Vittorini

Il Coordinatore del Dottorato

Prof. Franco Garofalo

Il Co-Tutore

Prof. Stefano Marrone

Dipartimento di Informatica e Sistemistica

Indice della Tesi

Indice delle Figure

Introduzione

I sistemi basati su elaboratore sono ormai presenti nella nostra vita di ogni giorno e ne influenzano molti aspetti, a volte senza che di ciò si abbia neanche precisa cognizione. I sistemi che forniscono supporto indispensabile alle attività economiche, ai trasporti e alle comunicazioni, per non tacere dei sistemi che vengono utilizzati in campo biomedico o nell'automazione industriale e di altri sistemi a cui sono demandati servizi critici in termini di sicurezza fisica di persone e beni, hanno ormai raggiunto livelli di complessità che ne rendono estremamente difficile la progettazione, lo sviluppo e - una volta operativi – la necessaria manutenzione al fine di garantire che i requisiti di affidabilità e sicurezza continuino ad essere soddisfatti durante l'esercizio.

Tali sistemi presentano in genere una architettura distribuita di grandi dimensioni in cui la complessità è dovuta anche alla forte eterogeneità dei componenti, sia a livello hardware che software. Essi devono rispondere a diversi requisiti, dettati nel caso di sistemi critici anche dagli standard internazionali, il cui soddisfacimento deve essere dimostrato, a volte anche ai fini della necessaria certificazione. Pertanto è indispensabile che il processo di sviluppo (esteso agli aspetti di manutenibilità) sia tale da rendere possibile l'evidenza formale del rispetto dei requisiti. A tale scopo sono stati investigati nel passato approcci di modellazione formale che, costruendo un modello del sistema, ne consentono l'analisi e quindi sia la validazione che la verifica di proprietà. E' ben noto però che l'applicazione di metodi formali in ambito industriale – sebbene fortemente consigliato quando non obbligatorio – è rallentato dalle difficoltà dettate da un lato dalla complessità dei modelli (che rispecchia la complessità dei sistemi), e dall'altro dalla necessità di avere personale formato nell'uso dei linguaggi formali e nello sviluppo di modelli. In altre parole, si ravvisa la necessità di ***metodologie di sviluppo e di strumenti di supporto alla modellazione***. Così come nel tempo si è affermata come disciplina l' ingegneria del software, così attualmente si sente la necessità di una *ingegneria dei modelli* che definisca le opportune metodologie e i processi necessari a ingegnerizzare le attività di sviluppo di modelli complessi e a migliorare il rapporto qualità/costo anche nell'ambito della modellazione formale.

Allo stato in letteratura sono presenti diversi approcci, che possono essere ricondotti a due linee fondamentali: a) metodologie e strumenti per la composizionalità ed il multiformalismo, e relative tecniche di soluzione; b) Derivazione (possibilmente automatica) di modelli formali da modelli di alto livello del sistema utilizzati nella specifica e nella progettazione.

La presente tesi si pone in questo contesto l'obiettivo di definire una metodologia di *sviluppo di modelli complessi* che consenta di integrare queste due linee e di trarre vantaggio quindi sia dalla possibilità di uno sviluppo di modelli "per componenti", sia dalla possibilità di derivare automaticamente modelli formali da modelli di alto livello del sistema

e dei relativi requisiti grazie alla attuale disponibilità di framework attualmente utilizzati nello sviluppo di sistemi software in ambito del Model Driven Engineering (MDE).

Il contributo originale della Tesi è nella definizione della metodologia e delle sue fasi, nella definizione e nello sviluppo di alcune tecniche a supporto e nel particolare focus applicativo orientato alla manutenibilità dei sistemi critici, aspetto questo non sufficientemente trattato in letteratura e di particolare importanza per i sistemi critici. Specificamente la tesi affronta le seguenti tematiche:

- Definizione di una metodologia per lo sviluppo e la generazione automatica di modelli formali di sistemi complessi basata su approcci e tecniche di MDE;
- Estensione del profilo MARTE-DAM (Modeling and Analysis Real Time Embedded Systems – Dependability Analysis and Modeling) per la specifica ad alto livello di sistemi affidabili al fine di poter esprimere concetti relativi alla manutenibilità – ed in particolare al fine di modellare specifici aspetti dei sistemi riparabili;
- Studio delle relazioni esistenti tra i linguaggi di specifica formali, con particolare riferimento alla relazione di tipo generalizzazione-specializzazione, al fine di ingegnerizzare il processo di generazione automatica di modelli espressi mediante diversi formalismi in relazione tra loro;
- Sviluppo di tecniche trasformazionali basate sui risultati sopra riportati per particolari classi di formalismi di interesse, in particolare sono state sviluppate opportune trasformazioni Model-to-Model/Model-to-Text per la generazione automatica di modelli basati su alberi dei guasti riparabili (RFT) a partire da una specifica MARTE-DAM ad alto livello del sistema.

I contribuiti proposti sono stati validati su sistemi critici reali e specificamente nell'ambito del trasporto ferroviario alla modellazione del Radio Block Centre, il cuore del sistema ERTMS/ETCS per l'alta velocità, confermando i benefici che possono essere ottenuti in termini di facilità ed efficienza nello sviluppo di modelli da un approccio automatico di generazione che – essendo basato su una specifica utilizzata nell'ambito del processo di sviluppo del sistema – risulta unificante ed estendibile, per poter essere impiegata – ad esempio – in contesti di generazione automatica dei test.

La tesi è organizzata come segue. Il capitolo 1 pone la tesi nel contesto dello sviluppo di modelli formali per l'analisi di sistemi critici, descrive le motivazioni per la definizione della metodologia proposta e presenta lo stato dell'arte relativo sia alla composizionalità ed al multiformalismo che alla generazione automatica di modelli formali.

Il capitolo 2 introduce i principi cardine su cui si fondano gli approcci model driven, ed evidenzia l'impatto del model driven engineering nell'ambito dello sviluppo dei sistemi.

Il capitolo 3 fornisce una descrizione dell'approccio metodologico proposto soffermandosi sulle diverse fasi che consentono di generare automaticamente un modello da una opportuna descrizione ad alto livello del sistema e dei requisiti che esso deve soddisfare (punto 1). Viene sottolineato il ruolo della profilazione UML e delle trasformazioni Model-to-Model/Model-to-Text nel passaggio da una fase all'altra. Viene descritto poi il processo di generazione automatica.

Il capitolo 4 istanzia la metodologia ed il processo descritti nel capitolo precedente, presentando al contempo i risultati descritti ai punti 2,3,4. In particolare si applica la metodologia a sistemi critici per affidabilità, e di tali sistemi si mettono in evidenza gli aspetti legati alla riparabilità, al fine di valutarne la disponibilità/indisponibilità a fronte di possibili interventi manutentivi e di diverse politiche di riparazione.

Il capitolo 5 introduce il caso di studio di riferimento (i.e. ERMTS/ETCS) e presenta l'applicazione della metodologia e delle tecniche sviluppate alla generazione automatica di un modello basato su Fault Tree Riparabili del sottosistema RBC, discutendo dei vantaggi che l'approccio proposto presenta nell'applicazione di tecniche di modellazione formale in contesti industriali.

Capitolo I
Lo Sviluppo di Modelli per l'Analisi di Sistemi Critici

1.1 Contesto

I sistemi critici sono definiti in letteratura come quei sistemi *"il cui eventuale fallimento nell'assolvere una o più delle funzioni ad esse richieste, può comportare conseguenze di notevole gravità in termini di vite umane e danni all'ambiente"* [Laprie]. Uno dei criteri in base al quale si possono classificare i sistemi critici è pertanto dettato dal tipo di conseguenze che derivano dal loro malfunzionamento:

- ***mission critical*:** sistemi il cui eventuale malfunzionamento provoca notevoli perdite in termini economici;

- ***safety critical*:** sistemi il cui eventuale malfunzionamento può comportare danni fisici a vite umane e all'ambiente circostante che, a sua volta, può provocare danni alle persone e/o altre forme di vita;

- ***security critical*:** sistemi il cui malfunzionamento non assicura l'integrità e la riservatezza dei dati gestiti.

Sistemi *safety critical* sono ad esempio quelli che controllano gli aerei, governano i sistemi di telecomunicazione, monitorano la nostra salute negli ospedali, o sostituiscono il lavoro umano in settori che, sempre di più, richiedono i massimi risultati nei minimi tempi. Paradigmatico di un sistema *security critical* è un sistema bancario, che consente di effettuare transazioni on line: eventuali alterazioni delle informazioni sensibili relative ai conti dei clienti dovute ad esempio ad intrusione da parte di hackers, comportano una violazione di riservatezza e di integrità con effetti anche disastrosi. Esemplificativo, invece, di un sistema *mission critical* può essere un sistema di navigazione di un veicolo spaziale senza equipaggio, la cui perdita non comporta danni in termini di vite umane ma sicuramente gravi danni in termini economici.

Il lavoro di Tesi è incentrato in particolare su quei sistemi critici (basati su calcolatore) per i quali i requisiti fondamentali sono l'affidabilità, la disponibilità, la manutenibilità e la sicurezza (*Reliability, Availability, Maintainability e Safety*, indicati nel seguito con l'acronimo *RAMS*). Secondo la definizione fornita in [Avizienis] i RAMS, insieme all'integrità ed alla riservatezza, sono tra gli *attributi* base di un sistema affidabile *(dependable*). ***L'affidabilità (dependability)*** è un requisito chiave per i sistemi critici, può essere definita come *"the ability to deliver service that can justifiably be trusted"* [Avizienis] e può essere descritta a partire da alcuni concetti fondamentali: le ***minacce***

(threats) quali guasti, errori e fallimenti; gli ***strumenti (means)*** quali la prevenzione, la tolleranza ai guasti, la rimozione delle minacce e dei relativi effetti, e gli ***attributi*** sopra menzionati. La differenza tra errore e fallimento risiede nel fatto che l'errore è un alteramento dello stato interno del sistema causato da un guasto, mentre il fallimento è dovuto al propagarsi dell'errore all'interfaccia del sistema (o di un sottosistema) in modo da provocare una alterazione percepibile del suo funzionamento.

Dunque la *dependability* è un concetto integrato in cui gli attributi RAMS giocano un ruolo fondamentale. La **Reliability** fornisce una misura della "continuità" del servizio offerto dal sistema, esprime cioè la capacità di un sistema di continuare a funzionare correttamente nel tempo sia in condizioni di normale funzionamento che in condizioni ostili o inaspettate. L' **Availability** è l'attributo che esprime la disponibilità del sistema nel fornire un servizio. Un suo indicatore è il tempo medio tra successivi fallimenti *(Mean Time Between Failures, MTBF)*, tempo durante il quale il sistema è quindi disponibile. La **Mantainability** è la capacità di essere sottoposto a modifiche e/o riparazioni. Un suo indicatore è il tempo medio tra successive azioni di riparazione *(Mean Time To Repair MTTR)*. Con riferimento alla manutenibilità, un sistema può essere sottoposto ad un attività di:

- *Recovery* ossia di ripristino del funzionamento con riparazioni che provvedono a rimuovere i guasti dopo la loro manifestazione (correct maintenance) o alla rilevazione e rimozione di guasti "dormienti "prima che essi si manifestino(preventive maintenance)
- *Re-configuration* ossia di ripristino mediante l'impiego di risorse ridondanti

Si parla inoltre di *adaptive o augmentive maintenance* a seconda che le modifiche siano volte ad adeguare il sistema ai cambiamenti dell'ambiente circostante o ad arricchirli con nuove funzionalità offerte.

Si noti infine che i termini Safety e Security, tradotti entrambi nella lingua italiana con il temine "sicurezza", esprimono in realtà concetti diversi: **Safety** è la sicurezza in termini di garanzia di incolumità di persone e ambiente, Security esprime il concetto di sicurezza in senso informatico (integrità e confidenzialità di dati), o – nell'ambito delle Infrastrutture Critiche – la protezione da minacce naturali o intenzionali. Un indicatore della Safety è il tempo medio tra l'occorrenza di successivi eventi pericolosi (*Mean Time Between Hazardous Event, MTBHE*).

La dimostrazione formale del rispetto dei requisiti RAMS da parte di un sistema critico è affidata ad un insieme di attività di **Verifica e Validazione (V&V**) regolate da standard internazionali. Tra le tecniche più largamente utilizzate per la V&V dei sistemi critici citiamo:

- L'iniezione dei guasti (*fault injection*) a livello hardware che può essere applicata in fase di verifica per valutare l'efficacia dei meccanismi di tolleranza ai guasti;

- L’ analisi statica, l’ispezione del codice ed il testing che consentono, in fase di verifica, la rilevazione di errori sistematici presenti nel codice prodotto;
- I modelli stocastici predittivi, utilizzati sia in fase di specifica che in fase di verifica a livello hardware;
- I metodi formali, utilizzati sia in fase di specifica che in fase di verifica al fine di fornire rigorosa evidenza della verifica di proprietà sia a livello software che a livello hardware.

Tali tecniche possono essere classificate come **simultative** (fault injection [Hsueh], software testing [Beizer]) o basate sullo sviluppo di **modelli formali** (metodi formali [Clarke]). I metodi simulativi si propongono di riprodurre il comportamento di un sistema reale e quindi di avere determinati output rispetto a determinati input. I metodi simulativi, però, pongono dei vincoli molto forti nell’ambito del processo di verifica e validazione di un prodotto:

- il testing entra molto tardi nel ciclo di vita di un prodotto, pertanto i costi del bug-fixing restano ancora elevati;
- per sistemi molto grandi l’ambiente di simulazione diventa spesso un prodotto a sé stante: ciò comporta, oltre ai costi aggiuntivi, il problema della “validazione” degli strumenti sviluppati;
- la simulazione non fornisce prova matematica esaustiva della correttezza del sistema.

L’ultimo fattore diventa estremamente importante per i sistemi safety critical quando non vengono individuati tutti i possibili azzardi e quindi, per quei particolari azzardi non testati, l’evoluzione del sistema stesso può provocarne il fallimento con possibili conseguenze per la vita umana.

In questa tesi ci focalizzeremo pertanto sulle tecniche di analisi basate sullo sviluppo di modelli formali che a differenza degli approcci simulativi possono valutare gli attributi della dependability sia in termini qualitativi che quantitativi, consentendo di ottenere sia la dimostrazione di proprietà che la valutazione di indici numerici, e contribuendo alla dimostrazione del soddisfacimento dei requisiti richiesti dagli standard internazionali, ma anche alla fase di progetto grazie alla possibilità di effettuare analisi di sensitività e supportare la scelta di parametri progettuali. Possiamo pertanto classificare i modelli per l’analisi dei sistemi critici in tre classi principali:

1) modelli per la verifica di proprietà, tra cui: i modelli basati su **algebre**, che permettono la modellazione dei sistemi come *transition system* caratterizzati da uno spazio degli stati, più o meno esplicitamente specificato, e da una serie di relazioni che esprimono le modalità di transizione da uno stato all’altro, *e* i modelli basati su **logiche**, che permettono di descrivere il comportamento e le proprietà dei sistemi attraverso la stesura di assiomi ed asserzioni. In

particolare in questa classe citiamo i modelli su cui è possibile applicare tecniche di *model checking o theorem proving* [Merz]: attraverso il model checking si valutano, nello spazio degli stati che caratterizza il modello del sistema, tutte le possibili tracce di esecuzione alla ricerca di quelle (i controesempi) che violano le proprietà specificate [Clarke]; attraverso il *theorem proving* [Duffy] si cerca di dimostrare che la specifica dei requisiti è una conseguenza logica della specifica (assiomatizzazione) dell'implementazione del sistema;

2) modelli che consentono la valutazione di indici quantitativi, come ad esempio **i modelli stocastici,** in particolare ricordiamo le Catene di Markov Tempo Continue (CTMC) [Ascher] che consentono di effettuare un certo tipo di analisi comportamentale e di manutenibilità del sistema;

3) **modelli basati su grafi**, che coniugano a volte entrambe le tipologie di analisi. Questi ultimi permettono una modellazione. Tra i formalismi "grafici" più utilizzati citiamo i seguenti:

- *Fault Trees (FT)* [FThandbook]: formalismo per la specifica dell'affidabilità di un sistema e, attraverso alcune loro estensioni tra cui i Repairable Fault Trees (RFT) [Codetta], anche delle caratteristiche di manutenibilità. I Fault Trees non hanno un elevato potere espressivo ma le tecniche di risoluzione dei modelli sono molto efficienti perchè basate su tecniche combinatorie che scalano al crescere delle dimensioni del sistema da sviluppare; sono molto utilizzati per la modellazione strutturale del sistema.
- *Timed/Stochastic Petri Nets* [Ajmone]: possono essere utilizzate per la modellazione della performability(prestazioni + affidabilità) di un sistema; ed inoltre consentono una modellazione dettagliata degli aspetti relativi alla manutenibilità. Una forte limitazione è costituita però dal ben noto problema dell'esplosione dello spazio degli stati, che rende difficile la risoluzione di modelli di grandi dimensioni;
- *Reti Bayesiane* [Charniak]: formalismo storicamente usato per la rappresentazione della conoscenza nell'ambito dell'intelligenza artificiale che le recenti ricerche applicano per la modellazione dell'affidabilità come strumento che bilancia il potere espressivo con l'efficienza di risoluzione ed utilizzato con successo nella modellazione dei sistemi affidabili, ad esempio in [Bobbio], [Delic].

Esistono molti altri formalismi utilizzabili in generale nell'analisi dei sistemi affidabili, o orientati all'analisi di particolari aspetti, ad esempio rispettivamente gli Automi Temporizzati e le Reti di Code, per non parlare delle diverse estensioni di formalismi (come accaduto per i Fault Tree e le Reti di Petri). Lo scopo di questo breve "excursus" certamente

non esaustivo è di fornire una base "culturale" alle riflessioni che motivano il lavoro di tesi e che sono espresse nella prossima sezione.

Il caso di studio di riferimento descritto nella presente Tesi sarà incentrato poi in particolare sull'analisi di disponibilità di un ***sistema di controllo*** *safety critical*, rappresentato dal sistema di gestione e controllo dei segnali in una rete ferroviaria. Un sistema di controllo è progettato per interagire con un ambiente reale, rispondendo in tempi brevi a stimoli esterni. Per tale motivo i sistemi di controllo sono anche detti ***sistemi reattivi*** e sono frequentemente sistemi ***real time***, dovendo cioè reagire a stimoli esterni in tempi prestabiliti: un fallimento catastrofico può verificarsi non solo se il sistema produce risultati non corretti ma anche se - pur fornendo risultati corretti- la risposta arriva "troppo tardi" [HEATH].

1.2 Motivazione

In generale dunque un sistema critico è tenuto a rispettare direttive di rigorosi standard internazionali che si pronunciano non solo sulle caratteristiche di cui deve godere il sistema stesso ma anche sui metodi da applicare nelle diverse fasi del ciclo di vita di tale sistema, che vanno dal design all'implementazione fino al testing. Nei diversi ambiti applicativi (avionico, ferroviario, medicale…) bisogna rispettare standard specifici, ad esempio [CENELEC1],[RTCA]. In particolare le norme CENELEC si riferiscono all'ambito dei sistemi ferroviari di cui ci occuperemo nello sviluppare il caso di studio nel Capitolo 5.
La sicurezza (safety) di tali sistemi è gestita a partire dalla definizione di indicatori, in particolare il SIL (*safety-integrity level*) che può assumere valori che vanno da 0 a 4 in ordine crescente di criticità. Questo valore non può però essere calcolato se non all'interno di un processo di sviluppo rigoroso. Questi sistemi sono in genere progettati e verificati secondo un processo di ciclo di vita. In Figura 1 è riportato il modello di sviluppo a "V" come specificato da uno degli standard RAMS per sistemi critici in [CENELEC1].
In tale schema è chiara la presenza di due fasi distinte evidenziate dalle frecce diagonali: la prima fase è legata alle attività di costruzione del sistema ed è caratterizzata da un livello di dettaglio sempre crescente; durante tutta questa fase sono effettuate attività di verifica volte a rispondere alla domanda "Stiamo sviluppando il sistema correttamente?", ossia: gli output della fase appena finita sono coerenti con i relativi ingressi?
Durante il secondo ramo del processo, vengono effettuate, a livelli di astrazione sempre maggiori, ossia partendo dal livello di implementazione del sistema fino alle sue caratterizzazione ingresso/uscita, fasi di verifica e validazione volte a determinare anche la risposta alla seguente domanda: "Stiamo sviluppando il sistema corretto?"

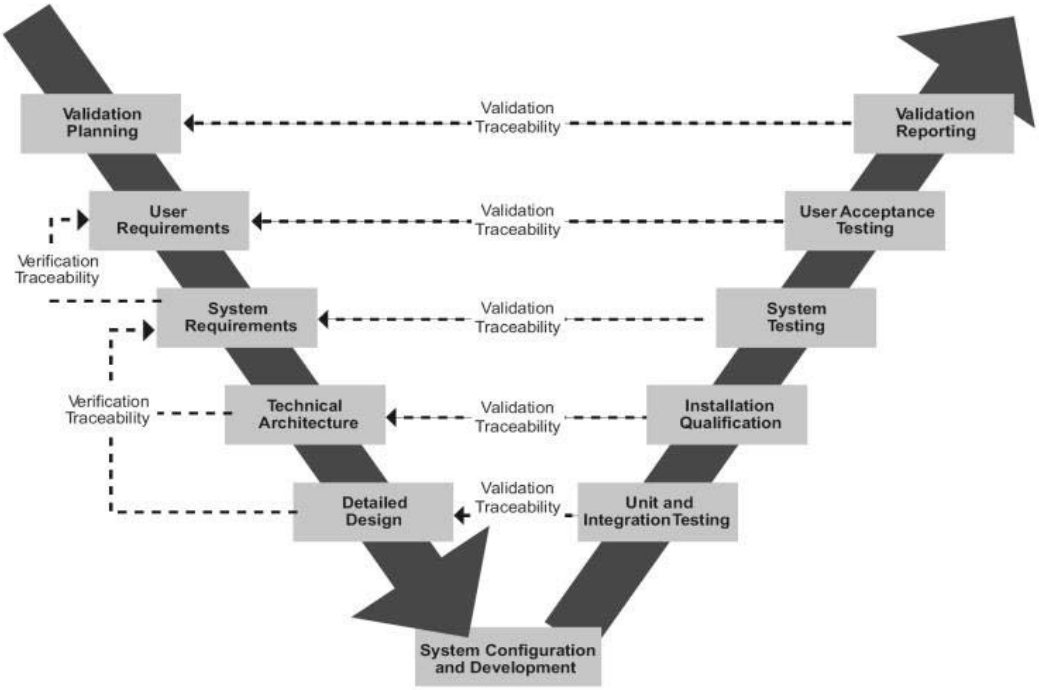

Figura 1: Ciclo di Sviluppo a "V" per Sistemi Critici di Controllo

Le attività V&V risultano però critiche sia in termini di budget che in termini di risultati per cui è strettamente necessario ricorrere a metodologie e strumenti che minimizzino il tempo richiesto per rendere operativo il sistema che si sta sviluppando, garantendo contestualmente il rispetto degli attributi RAMS espressi in fase di specifica.

Bisogna osservare però che le attività di V&V sono effettuate sia sull'hardware che sul software, inizialmente separatamente, finché non viene effettuato il test di integrazione ed il test del sistema.

A tale proposito bisogna dire che per quanto riguarda in particolare l'hardware, la probabilità che un guasto induca errori software è molto bassa, grazie alla applicazione di opportuni meccanismi di tolleranza ai guasti e che la validazione dell'hardware verso i requisiti di sicurezza (safety) è basata su tecniche ormai assestate. In altre parole la probabilità che si verifichi un fallimento hardware che possa portare il sistema in uno stato non sicuro ha raggiunto valori ampiamente accettabili, anche per gli standard internazionali.

Diversa è la situazione se si considera invece la validazione dell'hardware verso la disponibilità ad alto livello del sistema. Possono infatti essere definite, in questo caso, modalità di fallimento non banali che richiedono una attenta analisi.

Valutare gli effetti di scelte decisionali, operate durante la fase di progettazione, sulla disponibilità dell'intero sistema da realizzare al fine di verificarne l'aderenza ai requisiti di dependability, indicati in fase di specifica, comporta notevoli conseguenze in termini di riduzione dei tempi di sviluppo e quindi dei costi.

La scelta ovvia, infatti, di realizzare il sistema utilizzando in ogni caso le componenti più affidabili disponibili è in alcuni casi inapplicabile, essendo troppo conservativa. Esiste il rischio concreto di sovrastimare l'affidabilità del sistema e sovradimensionare i componenti utilizzati. Utilizzare le misure sul campo a partire da una reale istallazione del sistema per definire i parametri opportuni significa d'altra parte intervenire ovviamente troppo tardi, o a costi eccessivamente alti. E' pertanto necessario sviluppare modelli di alto livello del sistema fin dalle prime fasi del suo ciclo di vita usando linguaggi di modellazione opportuni che offrano un trade-off tra potere espressivo ed efficienza di risoluzione. I candidati migliori e raccomandati dagli standard internazionali, in questo ambito, sono i metodi formali che consentono di avere informazioni sulle proprietà (nel nostro caso di disponibilità) di un sistema fin dalle prime fasi del ciclo di vita del sistema da realizzare, riducendo la probabilità di dover correggere errori di progettazione individuati troppo tardi, a valle della fase di implementazione. I metodi formali forniscono *infatti "…mathematically based techniques that describe system properties. As such, they present a framework for sistematically specifying, developing and verifying systems"* [Wing]. Essi sono impiegati in diversi ambiti industriali, dallo sviluppo di microprocessori alla progettazione di sistemi software e di sistemi ibridi (hardware e software), come descritto in [Clarke]. Ciò nonostante la loro effettiva applicabilità soffre di alcune forti limitazioni dovute alla difficoltà di raggiungere il trade-off richiesto: da un lato, infatti, è necessario tener conto della mancanza di efficienza nella risoluzione di modelli complessi basati su un singolo formalismo, specialmente quando si debbano modellare anche le dipendenze tra i componenti e politiche di manutenzione che evidentemente hanno un significativo impatto sulla disponibilità complessiva del sistema, dall'altro – nel caso si opti per soluzioni più efficienti- dell'insufficiente potere espressivo del formalismo utilizzato. In aggiunta, lo sviluppo di modelli complessi è una attività che richiede specifiche competenze nell'utilizzo dei formalismi e nella modellazione dei sistemi ed anche propri tempi di sviluppo. La ricerca nell'ambito della modellazione formale dei sistemi in tempi recenti ha cercato quindi di proporre nuove ***metodologie di sviluppo e di strumenti di supporto alla modellazione*** con l'obiettivo di ingegnerizzare le attività di sviluppo di modelli complessi e migliorare il rapporto qualità/costo anche nell'ambito della modellazione formale.

Per queste ragioni il presente lavoro di Tesi si propone di fornire un contributo sia metodologico che tecnico a partire dalle due principali direzioni descritte in letteratura per affrontare i problemi derivanti dalla complessità dei modelli e dalla complessità del loro processo sviluppo: a) composizionalità ed il multiformalismo; b) Derivazione (possibilmente automatica) di modelli formali da modelli di alto livello del sistema utilizzati nella specifica e nella progettazione. Nel prossimo Capitolo viene descritto lo stato dell'arte relativo, con un accenno anche ad altre soluzioni proposte.

Uno degli obiettivi che si propone la presente Tesi è l'integrazione di queste due linee di ricerca e a tale scopo è definita la metodologia descritta nel Capitolo 3. I metodi e le tecniche sviluppati per definizione di un processo di sviluppo a partire dalla metodologia proposta sono presentate nel Capitolo 4. I contributi presentati nella Tesi sono poi validati applicandoli al caso di studio già menzionato nel contesto ERTMS-ETCS (European Railway Traffic Management System/European Train Control System), uno standard europeo applicato in Italia alle linee ferroviarie dell' "Alta Velocità" [UIC], [UNISIG]. Per i requisiti RAMS ci riferiremo ovviamente alle norme CENELEC [CENELEC1], [CENELEC2] per le apparecchiature di segnalamento ferroviario, le quali richiedono esplicitamente l'utilizzo dei metodi formali per la dimostrazione del soddisfacimento dei requisiti e la stima dei relativi indici.

1.3 Stato dell'Arte

Lo sviluppo di modelli per l'analisi di sistemi affidabili è una attività che richiede di affrontare numerose difficoltà, legate come si è detto principalmente: a) alla inadeguatezza del potere espressivo dei formalismi, nel momento in cui si devono modellare sistemi e comportamenti sempre più complessi e b) alla inadeguatezza delle tecniche risolutive dei modelli. Le soluzioni proposte dalla comunità scientifica al problema dello sviluppo di modelli complessi negli ultimi anni possono essere riportate essenzialmente alle seguenti direzioni di ricerca:

1. Approcci di tipo divide-et-impera, focalizzati sulla definizione di metodi e tecniche per uno sviluppo ***composizionale*** dei modelli. Un modello è costituito da più sottomodelli composti mediante opportune regole ed operatori di composizione. Parallelamente all'approccio composizionale è necessario evidentemente fornire appropriati metodi per la risoluzione e l'analisi dei modelli composti. Tali metodi devono comunque utilizzare:
    - tecniche per la limitazione delle dimensioni dello spazio degli stati (***largeness avoidance techniches***) che consentano di creare, ad esempio, rappresentazioni equivalenti ma di dimensioni inferiori, o di innalzare il livello di astrazione.
    - tecniche per la gestione delle dimensioni dei modelli (***largeness tolerance techniques***) che supportino la generazione di risoluzione di modelli di grandi dimensioni e di migliorare i tempi di risoluzione.

2. Approcci ***multiformali e multi-paradigma***, che esplorano la possibilità di combinare diversi formalismi e paradigmi di modellazione nella definizione del modello complessivo del sistema. In un certo senso questi approcci sono l'estensione naturale degli approcci di tipo divide-et-impera, nella direzione di "incapsulare" l'eterogeneità dei formalismi all'interno della definizione dei sottomodelli e di demandare ad

opportune tecniche e operatori di composizionalità l'integrazione tra sottomodelli e stili di modellazione differenti. Un interessante esempio di modellazione multi-paradigma è realizzato da AToM^3 [deLara] che implementa un approccio di modellazione basato sul meta-modelling e su tecniche di trasformazione tra grafi. I modelli possono essere traslati da un formalismo all'altro grazie ad un'apposita grammatica sui grafi, che definisce le regole di trasformazione. L'analisi dei modelli può essere effettuata sia attraverso tecniche di tipo simulativo che traslando i componenti in un formalismo comune ed applicando successivamente il solver specifico.

3. Approcci per la ***generazione (automatica) di modelli*** per l'analisi di affidabilità a partire da modelli di alto livello sviluppati nell'ambito di ambienti per il Model Driven Engineering.

Nella presente Tesi, si farà uso di alcuni approcci multiformali e si proporrà di integrare tali approcci in un processo di generazione automatica basato su metodi di Model Driven Engineering. Pertanto nel seguito ci si soffermerà, in particolare, su alcuni risultati descritti in letteratura nell'ambito del multiformalismo, mentre il prossimo capitolo è interamente dedicato al rapporto tra MDE e la modellazione di sistemi affidabili.

1.3.1 Il Multiformalismo nella Modellazione dei Sistemi Critici

Non sempre un singolo formalismo riesce a catturare le caratteristiche di sistemi critici complessi, per i quali l'analisi di affidabilità richiede che siano utilizzati opportuni linguaggi di modellazione, bilanciando potere espressivo e efficienza del processo di soluzione. Una soluzione "intuitiva" quindi è introdurre l'utilizzo di più linguaggi di modellazione in un approccio composizionale, in modo da poter opportunamente modellare componenti eterogenee o uno stesso sistema a differenti livelli di astrazione utilizzando il formalismo più indicato. Ciò richiede evidentemente sia uno sforzo teorico, che fornisca le basi per l'integrazione e l'interoperabilità tra formalismi diversi, sia le definizione e lo sviluppo di ambienti e strumenti per la modellazione e la risoluzione dei modelli multiformali.

I primi passi in questo senso sono stati compiuti dal progetto SMART [Ciardo] che integra i seguenti formalismi: Stochastic Petri Net, Catene di Markov Tempo Continuo e Tempo Discreto in un ambiente unico di modellazione, e da SHARPE [Trivedi], uno strumento per l'analisi di reliability e performability dei modelli che integra diversi formalismi quali i Fault Tree, Generalized Stochastic Petri Net, alcuni tipi di Reti di Code e Processi Markoviani. Sono stati poi proposti in letteratura alcuni approcci e alcuni strumenti esplicitamente basati su metodologie per lo sviluppo di modelli multiformali. I principali framework in questo senso sono Möbius [Deavours] e OsMoSyS [Vittorini].

Möbius vuole superare le limitazioni dei precedenti framework, come SMART e SHARPE, che consentono di comporre i modelli multiformali solo mediante scambio di risultati a valle dell'analisi dei singoli sottomodelli. Tramite Möbius infatti i modelli possono "interagire" grazie alle speciali funzionalità del motore di risoluzione, che consente, ad esempio, di scambiare anche risultati "parziali" durante le simulazioni. Di contro, per ottenere questo risultato è necessario che i modelli siano descritti – a partire dai diversi formalismi su cui si basano- in una notazione astratta comune, cosa che può portare ad una scarsa efficienza nel processo risolutivo. La scelta dei possibili formalismi da utilizzare resta, comunque, confinata ad un insieme di formalismi "compatibili": una assunzione fondamentale in Möbius è che il framework può operare solo con sistemi ad eventi discreti.

OsMoSys (Object-based multi-formaliSm MOdelling of SYStems) [Vittorini] nasce per fornire una metodologia e un ambiente per lo sviluppo di modelli multiformali che sia flessibile, estensibile e consenta l'integrazione di diversi motori di risoluzione in modo che essi risultino debolmente accoppiati. OsMoSys privilegia cioè la facilità di integrazione di nuovi formalismi all'interno del framework e l'orchestrazione dei solutori.
La relazione di composizione è un punto cruciale nella definizione di metodologie di modellazione avanzate. Essa deve tener conto non solo dell'integrazione dei differenti linguaggi di modellazione, in cui sono espressi gli stessi sottomodelli, ma anche dell'integrazione di ciascuno dei diversi tool con cui si possono risolvere i singoli sottomodelli. Al fine di realizzare una composizione corretta, un modello multiformale deve verificare determinate proprietà:

- non deve perdere traccia delle proprietà del sistema modellato e deve conservare le proprietà già verificate dai sottomodelli prima della stessa composizione;
- deve integrare i formalismi in maniera semanticamente corretta (per esempio, deve aver senso utilizzare la frequenza di scatto di una transizione di una Petri Net per valutare la probabilità di occorrenza di un evento in un Fault Tree);
- deve essere definito opportunamente il processo di risoluzione, in termini di flusso di dati e di esecuzione delle opportune attività che contribuiscono alla risoluzione del modello complessivo (per esempio, se il modello basato su Generalized Stochastic Petri Net(GSPN) [Ajmone] deve essere risolto prima del modello FT) e le operazioni da effettuare (per esempio, valutare il *throughput* di una transizione del modello GSPN, calcolarne l'inverso, usare questo dato come input di un parametro del modello FT).

Una volta risolte le su menzionate problematiche, che dipendono dai formalismi in considerazione e vanno affrontate nello specifico caso per caso, il multiformalismo risulta essere uno strumento molto potente perché consente di introdurre nell'ambito della

modellazione di sistemi complessi tecniche simili a quelle utilizzate nell'ingegneria del software nello sviluppo per componenti.
La metodologia proposta dal framework OsMoSys distingue due tipi di multiformalismo, che può essere *esplicito o implicito*.
Il **multiformalismo esplicito** vede la composizione di sottomodelli ciascuno dei quali è basato su un singolo linguaggio di modellazione. Il **multiformalismo implicito** vede la composizione di sottomodelli espressi mediante formalismi in cui almeno uno degli elementi del linguaggio nasconde l'utilizzo di un diverso formalismo. Il modello è apparentemente basato su un unico linguaggio, ma la sua risoluzione richiede l'utilizzo di tecniche proprie del multiformalismo. Un esempio è dato dai Fault Tree Riparabili (RFT) che utilizzeremo nello sviluppo del caso di studio e che pertanto vengono di seguito descritti in maggior dettaglio.

1.3.2 Multiformalismo Implicito: Fault Tree Riparabili

Un sistema riparabile è un sistema il cui corretto funzionamento, a seguito di un fallimento, può essere ripristinato attraverso una appropriata azione di manutenzione, piuttosto che mediante la sostituzione del sistema stesso [Ascher]. Gli attributi più importanti nella valutazione dell'affidabilità di un sistema riparabile sono la Reliability, la Maintainability e la Availability. L'Availability tra queste è la principale perché fornisce una misura sintetica delle proprietà di Reliability e Maintainability del sistema. Poiché è realistico assumere che il tempo necessario al prodursi di un fallimento del sistema dipenda dall'architettura del sistema stesso e dai parametri connessi alla Reliability, e non dal tempo trascorso dall'ultimo fallimento, si può supporre che la relativa distribuzione goda della proprietà di "memoryless" e sia pertanto descrivibile mediante variabili distribuite esponenzialmente. Qui assumiamo pertanto una distribuzione esponenziale sia per i tempi di riparazione che per i tempi intercorrenti tra successivi fallimenti. Nel caso in cui entrambe le distribuzioni siano tempo-invarianti può essere valutata una misura dell'Availability del sistema in condizioni di equilibrio (steady-state). Tale indice può essere ottenuto come:

MTTF/(MTTF+MTTR)

dove MTTF è la media statistica della variabile casuale "Time to Failure" (rispettivamente: "Time to Repair") [Barlow]. Si noti che l'MTTF è un valore caratterizzante le specifiche di un componente (gli attuali sistemi sono costruiti utilizzando sempre più frequentemente componenti COTS) riportato sui relativi data-sheet descriventi le caratteristiche tecniche e espresso equivalentemente come MTBF (Mean Time Between Failures) misurato in ore (h). L'analisi di Reliability di sistemi riparabili può essere condotta mediante modelli basati su Fault Trees o mediante Reliability Block Diagrams, entrambi molto utilizzati perche presentano alcuni indubbi vantaggi (sono formalismi intuitivi, facili da utilizzare, ed

efficienti nell'analisi dei modelli che è basata su tecniche di tipo combinatorio), ma come già accennato, poveri nel potere espressivo. Inoltre con questi formalismi non è possibile valutare politiche di riparazione [Cassady] che vengono più facilmente modellate, ad esempio, mediante le CMCT e le Generalized Stocastic Generalizzate Petri Nets(GSPN) [Ross], [Ajmone]. I Fault Tree Riparabili (RFT) [Codetta] sono stati introdotti proprio al fine di coniugare i vantaggi dei Fault Trees con il potere espressivo delle GSPN, allo stesso tempo consentendo -ove possibile- un efficiente processo di soluzione basato su un approccio divide-et-impera.

Gli RFT sono dunque una estensione dei Fault Trees ottenuta aggiungendo un nuovo tipo di nodo, il ***Repair Box (RB)*** al formalismo base. ***Un RB incapsula al suo interno un modello GSPN*** che modella qualunque tipo di intervento di manutenzione, consentendo di valutare l'effetto di politiche di riparazione anche complesse sulla Availabiliity del sistema. I modelli RFT vengono analizzati mediante un processo iterativo, a partire dagli eventi base (Basic Events) del Fault tree verso la radice.

Un Repair Box può rappresentare diversi aspetti di una azione di repair. In particolare:

- ***l'evento trigger***, cioè la condizione di guasto che determina l'intervento (l'evento del FT connesso al Repair Box);
- ***la politica di repair*** –unitamente alla procedura di repair, ai tempi richiesti, alle priorità e al numero di risorse disponibili per effettuare la riparazione, anche in termini di personale tecnico (modellata dalla GSPN);
- ***l'insieme dei componenti del sistema coinvolti nell'azione di repair*** (sottoinsieme dagli eventi foglia del sotto-albero la cui radice è l'evento trigger).

Graficamente, un RFT altro non è che un FT con l'aggiunta dei RBs e dei relativi archi. Un RB, infatti, è collegato all'albero mediante due tipi di archi: un arco che connette il RB al suo evento trigger, e gli archi che collegano il RB agli eventi foglia (i basic event) del Fault Tree sui quali il RB agisce (cioè gli archi che collegano il RB ai componenti del sistema che devono essere riparati).

Un modello RFT viene costruito in due passi. Inizialmente si costruisce il modello FT del sistema mediante una ispezione della sua struttura, poi si applicano al modello le politiche di repair identificando gli eventi trigger e il sottoinsieme di componenti ai quali tali politiche devono essere applicate.

Sotto opportune ipotesi di indipendenza degli eventi foglia, i sotto-alberi sono iterativamente risolti *traducendoli* secondo i diversi casi in:

- Fault trees, se ad essi non sono connessi ad un RB (i.e. non sono riparabili)
- una GSPN, se la loro radice è connessa ad un RB. Specificamente, il sottoalbero viene tradotto in una rete GSPN equivalente, mediante le regole di traduzione definite in [Bobbio2] e quindi composto con la rete GSPN incapsulata nel RB. L'effetto della politica di repair definita dal RB sulla parte del sistema coinvolto, viene così valutata, il sotto-albero viene sostituito da un unico evento foglia, la cui probabilità di occorrenza è ottenuta

utilizzando i risultati dell'analisi. Una volta ottenuto un albero in cui tutti i RB siano stati "risolti", possono essere utilizzate le tecniche combinatorie classiche per valutare l'Availability del sistema a fronte delle politiche di repair. In questo modo il potere espressivo delle GSPN è preservato, ma la dimensione del modello GSPN è ridotta, e parte del modello può ancora essere risolto mediante tecniche combinatorie.

1.3.3 Generazione Automatica di Modelli Formali

La generazione di modelli formali per l'analisi dell'affidabilità a partire da modelli UML del sistema è una strada già percorsa dalla comunità scientifica, ad esempio in [D'Ambrogio] è descritto come ottenere un modello FT da un insieme di diagrammi UML, in [Bondavalli1] diagrammi UML descriventi la struttura del sistema sono utilizzati per derivare mediante opportune trasformazioni modelli basati su Timed Petri Nets, mentre diagrammi comportamentali (i.e. statecharts con guardie sugli archi) sono utilizzati in [D'Ambrogio], una trasformazione da Statechart verso Stochastic Reward Nets è presentata in [Huszerl]. In letteratura sono stati proposti anche alcuni strumenti di supporto alle tecniche trasformazionali citate, ad esempio, OpenSESAME [Walter] consente di generare Stochastic Petri Nets a partire da rappresentazioni grafiche di alto livello che descrivono dipendenze tra le componenti del sistema.

Questi e altri risultati descritti in letteratura hanno evidenziato i vantaggi che possono derivare da approcci basati su tecniche trasformazionali a partire da modelli tipicamente utilizzati nell'ingegneria del software, ed hanno aperto la strada, da un lato, ad una intensa attività di ricerca finalizzata alla adozione di principi MDE nella analisi di affidabilità dei sistemi mediante la definizione di trasformazioni *dirette* dal modello architetturale di alto livello ai modelli di analisi, dall'altro alla definizione di linguaggi di modellazione e di estensioni degli attuali linguaggi (principalmente profili UML) al fine di fornire ad alto livello gli elementi linguistici necessari alla modellazione di proprietà non funzionali e di concetti relativi all'analisi di affidabilità dei sistemi, quali ad esempio il profilo OMG QFTP [OMG_QFTP]. Uno dei profili più interessanti in questo contesto è il profilo DAM [Bernardi], un profilo UML specifico per l'analisi dell'affidabilità basato su MARTE [OMG_MARTE], che a sua volta consente di modellare sistemi embedded real time.

Nella maggior parte dei lavori basati su principi MDE, descritti in letteratura, però, le trasformazioni non sono integrate in un processo di sviluppo più generale, e pertanto tali approcci soffrono di due principali limitazioni:

1) La definizione delle trasformazioni è una attività non sufficientemente "ingegnerizzata", legata spesso a domini applicativi molto specifici, risultante quindi in regole di trasformazione poco *flessibili e non riusabili;*

2) L’intrinseca complessità dell’analisi di affidabilità, come discussa nelle precedenti sezioni, non è sufficientemente affrontata. Non esiste di fatto un framework generale nel quale gli approcci trasformazionali siano integrati/integrabili con gli approcci sopra descritti per la gestione della eterogeneità di proprietà, metodi di analisi e strumenti necessari ad affrontare le sfide poste dallo sviluppo di modelli per sistemi critici.

Per quanto concerne la ingegnerizzazione delle trasformazioni, alcuni lavori recenti affrontano la definizione di processi scalabili per lo sviluppo di trasformazioni tra modelli (ad esempio in [Kurtev], [Balogh1] e [Balogh2]). Questi lavori presentano una panoramica su alcuni metodi per la definizione di regole di ereditarietà tra trasformazioni prevalentemente orientate all'ereditarietà di regole di trasformazione. Sull’altro fronte, la necessità di sviluppare opportune metodologie è stata raccolta da alcuni progetti, tra cui i progetti HIDE [Bondavalli2], PRIDE [Pride], e più recentemente il progetto europeo SATURN (SysML based modelling, architecture exploration, simulation and synthesis for complex embedded systems” [Saturn]) ed il progetto ARTEMIS-JU CHESS (“Composition with guarantees for High-integrity Embedded Software components aSsembly”, [Chess] [Montecchi]) il cui obiettivo è la creazione di framework per l'analisi di sistemi critici mediante approcci model driven che supportino l’intero ciclo di sviluppo.

In questa direzione si colloca anche il presente lavoro di Tesi.

Capitolo II
Model Driven Engineering nei sistemi critici

2.1 Model Driven Engineering: una introduzione

Uno dei più recenti trend di ricerca nell'ambito dell'ingegneria del software è il Model Driven Engineering (MDE) [Schmidt] nato come estensione di approcci più specifici come il Model-Driven Architecture (MDA) [Mellor] del Object Management Group. Gli approcci del MDE si pongono come obiettivo la definizione di metodologie e tecniche a supporto dei processi relativi all'intero ciclo di vita di sviluppo di un software (sistema) attraverso la manipolazione (automatica) di modelli.

Prima di proseguire ulteriormente, occorre chiarire la differenza tra Model Driven Architecture, Model Driven Software Developement (MDSD) e Model Driven Engineering. Il primo è, a tutti gli effetti, uno standard OMG, focalizzato allo sviluppo software che prescrive in modo " mandatory" quelli che sono i linguaggi usati allo scopo (ovviamente UML in quanto anch'esso standard OMG); per quanto riguarda MDSD, il focus è sempre quello dello sviluppo software, ma in maniera indipendente da linguaggi specifici. Il Model Driven Engineering non solo fa cadere la restrizione legata al contesto software ma si slega anche da particolari processi di sviluppo (è possibile dunque definire anche processi model driven di reingegnerizzazione o altro...).

Al centro di un qualsiasi processo MDE c'è il modello. Nel nostro contesto, un modello è tale quando il linguaggio di modellazione in cui viene espresso è caratterizzato da una sintassi ed una semantica ben formalizzate, condizione necessaria per la trasformazione automatica dei modelli stessi. Il modello rappresenta il sistema e ne costituisce un formalizzazione astratta e conforme ad un determinato linguaggio. La potenza dell'approccio model driven consiste anche nel considerare un linguaggio come una sorta di "sistema", anch'esso modellabile attraverso una descrizione formale ed astratta: il meta-modello. Questo concetto si lega con quello di modello di dominio: ***il modello deve rappresentare un'astrazione dei concetti del sistema da realizzare rispetto ad un determinato dominio applicativo*** anch'esso rappresentato da un (meta-)modello. Questo ragionamento può essere ulteriormente iterato, fino a definire un modello per la rappresentazione di linguaggi usati a loro volta per formalizzare altri linguaggi (il meta-meta-modello). Gli approcci model driven sono quindi storicamente definiti sulla base di "pile" di modelli come raffigurato in Figura 2, dove è mostrato lo stack relativo ai linguaggi di modellazione dell'OMG.

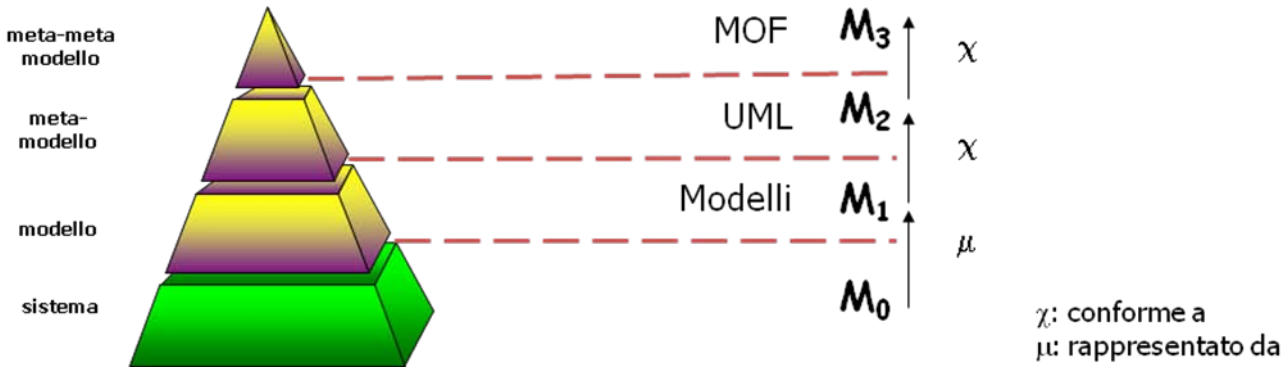

Figura 2: Stack dei modelli - MOF

Una delle tecniche principali utilizzate in ambito MDE è la generazione automatica di modelli sulla base della definizione di trasformazioni automatiche definite sui linguaggi. In Figura 3 è raffigurata una trasformazione, una volta definiti sia un linguaggio di partenza che di destinazione, che consente di trasformare un qualsiasi modello conforme al primo linguaggio, in un nuovo modello conforme al secondo.

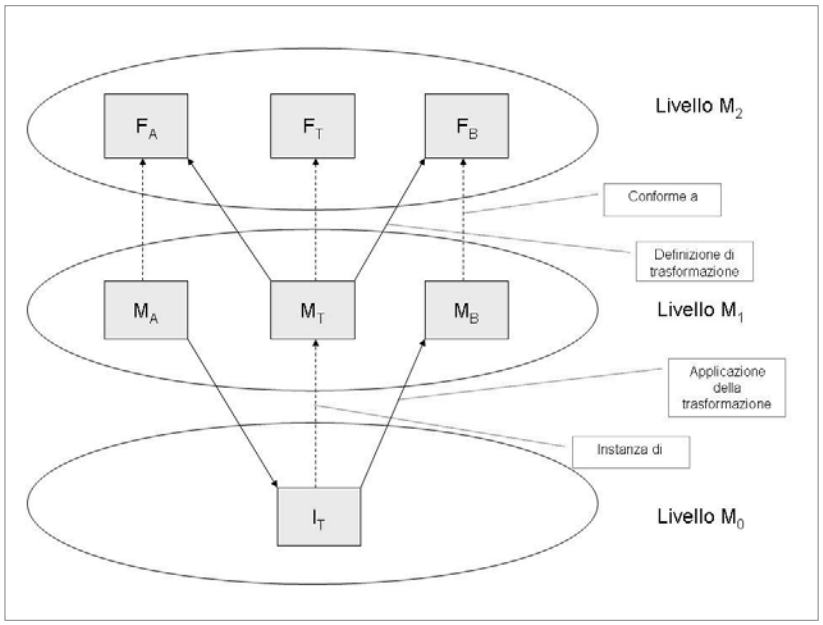

Figura 3: Trasformazioni tra modelli - schema generale

Formalizzando quanto appena detto, possiamo affermare che due sono i principi cardine su cui si basa l'approccio MDE:

- ***la creazione di modelli***, basata sulla definizione di modeling stacks mediante tecniche di meta-modelling, dove i modelli di livello inferiore si conformano a quanto specificato in uno o più modelli di livello superiore;

- ***la manipolazione dei modelli***, mediante trasformazioni che generino, a partire da modelli sorgente (input della trasformazione) conformi a meta-modelli sorgente, modelli destinazione (output della trasformazione) conformi a meta-modelli destinazione.

MDE consente di gestire la complessità di un sistema garantendo un aumento di produttività da una parte ed il miglioramento della qualità dall'altra, promuovendo un maggior livello di astrazione e riuso. Ciò viene fatto mediante un processo di ingegnerizzazione, sia dei linguaggi che delle trasformazioni: esse sono alla base del processo di sviluppo di un sistema perché rappresentano la modalità di passaggio da un modello iniziale di più alta astrazione ad un modello più concreto. Dunque un processo model driven relativo ad un sistema può essere concepito come una catena di trasformazioni in grado di trasformare il modello iniziale in un modello finale.

2.2 MDE e linguaggi

Nel contesto MDE costituisce quasi disciplina a sé la definizione dei linguaggi cosiddetti di "Dominio": i ***Domain Specific Modeling Languages (DSML),*** ossia i linguaggi di modellazione specifici di dominio. Un DSML altro non è che un linguaggio in grado di catturare concetti non presenti in un linguaggio "generalista" quale UML o Ecore: esso pone l'accento su un particolare dominio applicativo. All'interno dei DSML possiamo distinguere essenzialmente due categorie: quella dei linguaggi orizzontali (anche detti technical-***oriented DSML***) e quella dei linguaggi verticali (anche detti ***business-oriented DSML***). I primi linguaggi sono quelli relativi a domini trasversali a tanti ambiti applicativi: uno di questi linguaggi è, ad esempio, quello relativo al testing o, come vedremo successivamente, MARTE. I linguaggi di tipo verticale si caratterizzano invece dall'avere una maggiore profondità all'interno dello specifico dominio applicativo (ferroviario, aerospaziale, etc...).

Il processo di creazione di un DSML parte da quella che è la definizione di un meta-modello che rappresenti i concetti del dominio (orizzontale o verticale che sia). Sulla base di tale meta-modello è poi possibile concretizzare il linguaggio o creando da zero il nuovo linguaggio o estendendone uno già esistente.

Il primo approccio è caratterizzato dall'ottenere dei risultati migliori in termini di "snellezza" del linguaggio in quanto è un approccio che garantisce la massima flessibilità nella scelta della sintassi (astratta e concreta) nel rispetto del meta-meta-modello di

riferimento scelto. Questo processo è, però, generalmente più costoso in quanto occorrerà generare da zero il nuovo linguaggio. Un approccio alternativo è fornito dalla possibilità di estendere linguaggi esistenti al fine di introdurre nuovi concetti precedentemente assenti. E' questo il caso del linguaggio UML che, attraverso il meccanismo della profilazione, consente la definizione di nuovi "moduli linguistici". Il vantaggio di questo ultimo approccio risiede, oltre che nella grande diffusione del linguaggio UML ed allo sconfinato supporto da parte di strumenti commerciali e non, nella maggiore rapidità con il quale possiamo generare nuovi linguaggi; d'altro canto il legarsi ad un linguaggio esistente, per quanto generalista come UML, riduce la flessibilità e quindi la capacità del linguaggio di adattarsi ad esigenze specifiche dell'utilizzatore del linguaggio stesso. Nel resto della tesi lavoreremo essenzialmente con profili UML.

Un profilo UML è essenzialmente costituito da tre elementi fondamentali:

- *stereotipo*: estensione UML delle meta-classi laddove con il termine meta-classe si indica l'elemento del meta-modello che si vuole estendere. E' una nuova classe battezzata con il termine <<stereotipo>> costituita da un nome e dalle meta-classi che si desidera classificare ad un livello più alto (rappresentandone il significato, l'utilizzo);
- *constraints*: per associare dei vincoli agli stereotipi si impongono delle restrizioni sulle meta-classi. Introducono regole che possono essere verificate nel modello di disegno fornendo un modo per esprimere requisiti;
- *tagged value*: sono caratterizzati da un nome e da un tipo e compaiono come attributi della classe stereotipo ossia come meta-attributi aggiuntivi delle meta-classi che si vogliono estendere, nella forma di una coppia 'tag=valore'.

2.3 MDE e Trasformazioni

In base all'approccio MDE, il sistema da realizzare è rappresentato lungo l'intero processo di sviluppo, dall'analisi dei requisiti alla progettazione fino alla fase di implementazione, da modelli, ognuno dei quali rappresenta ad un diverso livello di astrazione ,via via sempre più dettagliato, il sistema da realizzare e si ottiene mediante una trasformazione del modello generato dalla fase precedente: si può pensare a ciascuna fase come ad una trasformazione del modello, che riceve in input, il modello generato dalla fase precedente e genera un nuovo modello che sarà l'input della fase successiva.

2.3.1 Le trasformazioni di modelli

Quando si parla di trasformazione è necessario capire qual è l'input su cui lavorerà e quale deve essere l'output che essa deve generare. Se come input e output della trasformazione si hanno dei modelli si parla di trasformazioni Model-to-Model (M2M). Quando come output

si ha del testo (formato necessario affinché risulti gestibile da un risolutore) si parla di Model-to-Text (M2T) ed in particolare quando il testo è costituito da codice sorgente scritto in un determinato linguaggio di programmazione si parla di trasformazione Model-to-Code (M2C).

Secondo una definizione di Kleppe e al. [Kleppe] ***una trasformazione è la generazione automatica di un modello destinazione a partire da un modello sorgente***.

Una trasformazione è, pertanto, definita da un insieme di regole che descrivono come un modello espresso nel linguaggio sorgente può essere trasformato in un modello espresso nel linguaggio destinazione. ***Una regola di trasformazione*** specifica come uno o più costrutti del linguaggio sorgente possono essere trasformati in uno o più costrutti del linguaggio destinazione.

In particolare trasformare un modello sorgente conforme ad un meta-modello sorgente in un modello destinazione (conforme ad un meta-modello destinazione) significa definire delle regole di trasformazione o "transformation rules" tra gli elementi del meta-modello sorgente e quelli del meta-modello destinazione. Dunque, dato un modello sorgente, è necessario avere a disposizione il meta-modello destinazione, in modo da poter definire un mapping tra i costrutti del primo e quelli del secondo. Una trasformazione di modelli può essere applicata a più modelli sorgente o generare più modelli di output.

Secondo una corrente di pensiero della comunità scientifica, le trasformazioni possono essere considerate come modelli a tutti gli effetti: ciò implica la necessità di far riferimento ad un linguaggio di trasformazione ben formalizzato e soprattutto la possibilità di applicare alle trasformazioni tutte le tecniche già consolidate nell'ambito dell'ingegneria del software.

Una trasformazione di modelli può essere definita ***verticale*** quando trasforma un modello più astratto in uno più concreto (ad es.il modello di disegno in codice) perché il modello sorgente e quello destinazione sono a due livelli di astrazione diversi, o ***orizzontale*** se il modello sorgente ed il modello destinazione sono caratterizzati dallo stesso livello di astrazione.

Quando il modello sorgente e quello destinazione di una trasformazione di modelli sono espressi nello stesso linguaggio di modellazione si parla di *traslazione di modello* o *riformulazione*, quando sono espressi in linguaggi differenti, di *traslazione di linguaggio* o semplicemente *traslazione*. Si parla anche di *trasformazioni endogene* nel primo caso e di *trasformazioni esogene* nel secondo.

Le riformulazioni si dividono a loro volta in:

- *Normalizzazione*, quando il modello destinazione è conforme ad un meta-modello che risulta un sottolinguaggio del linguaggio di modellazione sorgente per ridurre la complessità sintattica;
- *Rifattorizzazione*, quando il modello di destinazione deriva da una ristrutturazione del modello sorgente (si modifica solo la struttura interna del software) in modo da

migliorarne la qualità interna (modificabilità, riusabilità) senza cambiarne il comportamento verso l'esterno;
- *Correzione*, quando la trasformazione è finalizzata ad eliminare gli errori;
- *Adattamento*, quando la trasformazione viene usata per adeguare il modello a nuove esigenze.

Le traslazioni di linguaggio si distinguono in:
- *Migrazione*, se i linguaggi di modellazione sorgente e destinazione sono diversi ma caratterizzati dallo stesso livello di astrazione;
- *Sintesi*, se si passa da un linguaggio di più alto livello, più astratto ad uno di più basso livello, più concreto. Un esempio di sintesi è la trasformazione da un modello di analisi in un modello di disegno e da un modello di disegno in codice di un programma (in un determinato linguaggio). Un altro tipico esempio è la generazione di codice ossia il passaggio dal codice sorgente in bytecode o codice eseguibile;
- *Reingegnerizzazione*, è la trasformazione inversa della sintesi e trasforma un modello di basso livello in un modello di più alto livello (il linguaggio di modellazione sorgente è meno astratto di quello destinazione).

2.3.2 Strumenti a supporto delle trasformazioni tra modelli

Una serie di requisiti che caratterizzano una trasformazione di modelli è indicata di seguito:
- **il livello di automazione**: alcune trasformazioni possono essere completamente automatizzate , altre, anche se solo in parte, hanno bisogno dell'intervento umano. Si pensi ad es. alla trasformazione da un modello di requisiti in un modello di analisi in cui l'intervento umano è necessario per risolvere ambiguità, incompletezze ed inconsistenze derivanti dal linguaggio naturale in cui sono parzialmente espressi i requisiti;
- **la complessità**: le trasformazioni meno complesse come la riformulazione(trasformazione endogena che modifica la struttura interna del software per migliorarne le caratteristiche qualitative quali comprensibilità, modificabilità, riusabilità, modularità e adattabilità senza modificarne il comportamento) richiedono un insieme di tecniche e tool completamente differente da quello necessario per le trasformazioni più complesse;
- **il tipo di proprietà che deve essere preservata**: ogni trasformazione conserva nel modello destinazione solo certi aspetti del modello sorgente. Le proprietà che vengono preservate dipendono dal tipo di trasformazione. Ad es. con la rifattorizzazione ciò che viene preservato è il comportamento, con il refinement (passaggio graduale dalla specifica all'implementazione), invece, è la correttezza del programma. Nel caso di una trasformazione di database è necessario preservare l'integrità del database, nel caso di una trasformazione M2C è necessario preservare la correttezza, il tipo e la proprietà di well-formedness della sintassi;

- **verifica formale della trasformazione**: uno dei maggiori problemi relativi alla costruzione e alla verifica dei sistemi attraverso approcci composizionali è legata alla fidatezza ed alla correttezza che le trasformazioni introdotte devono possedere.

Al fine di scegliere un linguaggio per definire una trasformazione di modelli ed un tool per il suo utilizzo in grado di esaminare il modello generato dalla stessa trasformazione è consigliabile prendere in considerazione una serie di fattori quali:

- la possibilità di suggerire quando applicare le trasformazioni, in quale contesto o quale tipo di trasformazione di modelli è più appropriata in un determinato contesto.
- la possibilità di adattare le trasformazioni a specifiche esigenze e riutilizzarle.
- In questo caso ulteriori requisiti che possono essere soddisfatti da linguaggi di trasformazioni e tool consistono nel:
    - offrire tecniche per testare e validare le trasformazioni e per assicurarsi che esse abbiano il comportamento atteso;
    - avere la capacità di fondere trasformazioni esistenti in una sola al fine di incrementare la scalabilità degli approcci e dei processi proposti e, dualmente, di decomporre una trasformazione complessa in più trasformazioni più semplici;
    - essere bidirezionali: ogni trasformazione è utilizzabile per trasformare il modello sorgente in modello destinazione e viceversa;
    - supportare la tracciabilità e la propagazione del cambiamento: linguaggi e tool devono fornire rispettivamente meccanismi per mantenere un link esplicito tra il modello sorgente e quello destinazione di una trasformazione di modelli. Ciò è necessario al fine di supportare la propagazione di un cambiamento all'interno dell'intera catena dei modelli.
- Oltre a tali caratteristiche è inoltre preferibile che gli strumenti a supporto della model transformation godano anche delle seguenti proprietà:
- essere non solo utili ma anche facilmente utilizzabili e intuitivi;
- garantire il giusto numero di costrutti linguistici (al fine di massimizzare contemporaneamente espressività ed usabilità);
- supportare trasformazioni complesse o modelli complessi senza compromettere le performance;
- garantire estensibilità ossia flessibilità nel caso di aggiunta di nuove funzionalità;
- garantire integrabilità con altri tool usati nel processo model driven di ingegnerizzazione del software;
- essere compatibili con i principali standard di rappresentazione dati e linguaggi (XML, MOF, UML, Ecore).

2.4 MDE e sistemi critici

Nella loro generalità, i metodi e le tecniche di model driven engineering possono essere applicate anche ai sistemi critici. In particolare, il ruolo fondamentale che le attività di verifica e validazione ricoprono all'interno dei processi di sviluppo di tali sistemi, deve indirizzare gli sforzi di formalizzazione ed automatizzazione che tali tecniche permettono in queste fasi stesse. I benefici sono grandissimi: il paradigma di progettazione di sistemi "construct-by-correction" tipico dei processi basati su testing e sulla verifica late-time delle proprietà di un sistema, può essere sostituito da paradigmi "correct-by-construction" dove, la costruzione (e la verifica) di un modello iniziale del sistema e la verifica delle trasformazioni automatiche comporta un modello finale (il sistema in altre parole) automaticamente corretto. Lo sforzo di verifica può essere dunque concentrato nelle fasi iniziali del ciclo di vita del prodotto (early-time).
All'interno dei processi di verifica e validazione precedentemente illustrati e all'interno dell'ambito delle tecniche e dei metodi propri del Model Driven Engineering, si definisce chiaramente un primo scenario di applicazione di tali tecniche ai sistemi critici. Tale scenario è costituito dalla possibilità di aumentare la diffusione dei metodi formali all'interno di ambienti industriali nei processi di sviluppo di sistemi reali. Infatti la possibilità di definire linguaggi di alto livello vicini all'utente (per astrazione e semplicità di utilizzo), nonché la possibilità di derivare automaticamente modelli (formali atti all'analisi) a partire dai primi, permettono un'applicazione "trasparente" all'utente di metodi formali.
D'ora in poi indicheremo con formali i linguaggi preposti alla valutazione delle proprietà del sistema (Petri Nets, Fault tree etc...). Tale è in realtà un abuso di notazione in quanto anche i linguaggi di alto livello necessari alla generazione di tali modelli attraverso model trasformation devono essere "formali"; in realtà si dovrebbe parlare di linguaggi formali orientati alla modellazione e linguaggi formali orientati all'analisi.
Nel resto del capitolo illustreremo i principali approcci alla modellazione di sistemi critici ed alla generazione automatica di modelli formali.

2.4.1 Linguaggi di specifica per sistemi critici

Come evidenziato nel primo capitolo, i sistemi critici si caratterizzano dall'avere stringenti requisiti non funzionali. Il primo passo è stato, dunque, quello di studiare i linguaggi per la specifica di proprietà non funzionali di sistemi (d'ora in avanti NFP - Non Functional Properties). In tale contesto, un ruolo centrale viene svolto dal linguaggio UML in quanto, innanzitutto, è ormai standard de facto nella modellazione e, in secondo luogo, rappresenta il linguaggio più facile ed usato per la creazione di nuovi linguaggi attraverso il già descritto meccanismo dei profili.

Per far fronte alla necessità di rappresentare requisiti e proprietà di tipo non funzionale, da ora in poi, sia di tipo quantitativo che di tipo qualitativo, è necessario dunque aumentare la potenza espressiva di UML introducendo nuove notazioni.
La tipologia dello specifico dominio di analisi determina il particolare tipo di **NFP** da associare alle meta-classi. In base al tipo di **NFP** associate si potrà effettuare un'analisi di schedulabilità, piuttosto che di performance o dependability. Un esempio, nel caso di analisi di performance, di "*Annotated Element*" è costituito da una Resource o da un Service (operazione offerta dalla Resource) caratterizzati dalle **NFP** quali il tasso di utilizzo della Risorsa, i tempi di risposta ed il throughput del Servizio. Le NFP sono modellate da grandezze *Quantities* esprimibili in formati numerici, che possono essere base o derivate e che caratterizzano il sistema modellato. Le NFP descrivono le modalità di comportamento del sistema che si sta utilizzando come, ad esempio, le performance, la memoria usata e l'elettricità consumata. Possono essere di due tipi: **NFPs quantitative** quando si tratta di quantità che possono essere misurate (per enumerazione, comparazione o in modo assoluto) attraverso valore numerici (come l'energia, la dimensione o la durata temporale) e **NFPs qualitative** qualora tali caratteristiche non possano esser misurate numericamente. Nelle prime il valore numerico ha un significato quando viene specificata anche l'unità di misura, mentre quelle qualitative sono descrittive specificando la tipologia (ad es. una frequenza può essere caratterizzata qualitativamente come sporadica o periodica) [Sebastien].
Il profilo più diffuso in tale ambito, in quanto oramai standard esso stesso, è **MARTE** (**M**odelling and **A**nalysis of **R**eal -**T**ime **E**mbedded **S**ystem), un profilo nato dall'evoluzione di quello SPT dedicato allo Scheduling, Performance and Time.

2.4.2 Sistemi Real Time (MARTE)

Il primo passo in tale direzione è stata l'introduzione di notazioni per proprietà non funzionali NFP in un modello UML [Espinoza] con l'obiettivo di rappresentare in un modello UML le NFP legate alla generica analisi di tipo quantitativo dando vita al profilo MARTE, in Figura 4.
Il profilo MARTE [OMG_MARTE] nasce da una volontà di unificare package e profili precedentemente proposti nell'ambito della descrizione delle NFPs di un sistema. Esso si struttura in quattro package. Il primo regola la creazione del modello del sistema real-time ed embedded in esame (*MARTE Foundations*), il secondo e il terzo specializzano gli aspetti di rappresentazione (*MARTE DesignModel*) e di analisi (*RTEA RealTime & Embedded Analysing*), mentre l'ultimo fornisce il linguaggio di specifica di MARTE e la libreria del modello (*MARTE annexes*).

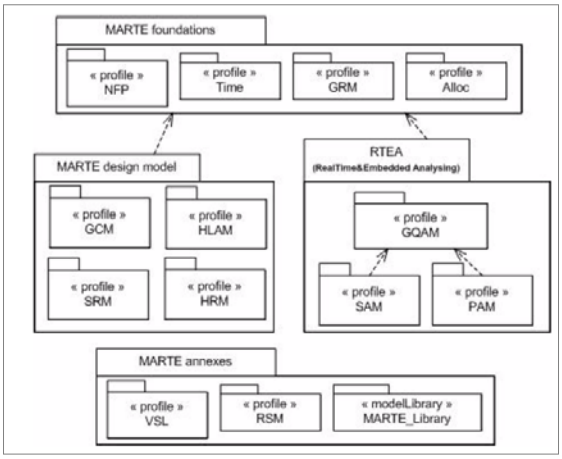

Figura 4: il profilo MARTE

Il package *MARTE Foundations* è mostrato in Figura 5 e contiene le definizione dei tipi elementari e degli stereotipi base da cui deriveranno tutti gli altri.

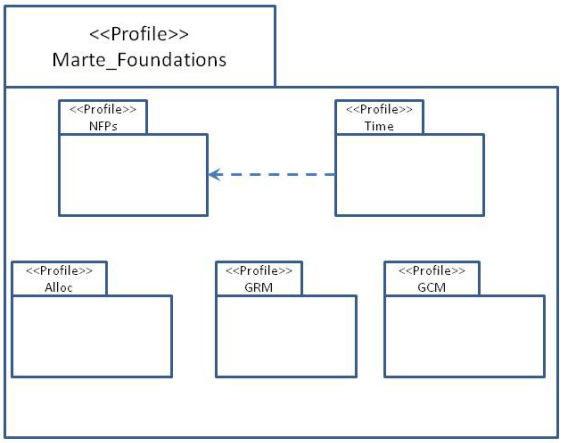

Figura 5: il Package MARTE Foundation

Il profilo definisce un insieme di concetti di base per i sistemi real-time e consiste di cinque sotto-profili. In particolare esso contiene il pacchetto *Time* molto importante nella definizione di sistemi real-time. MARTE, infatti, pone particolare attenzione a questo pacchetto e adotta modelli di tempo che dipendono dall'ordinamento parziale degli istanti. In particolare abbiamo tre modelli di base del tempo: *Chronometric*, *Logical* e *Synchronous*. Il primo riguarda principalmente la cardinalità del tempo (per esempio il ritardo, la durata ed il tempo di clock), il secondo gli eventi ordinati (per esempio si occupa

di descrivere che l'evento 1 accada prima del 2) ed il terzo è una specializzazione del modello del tempo logico e introduce il concetto di simultaneità (ad esempio che l'evento 1 accada nello stesso istante dell'evento 2).
Un altro pacchetto fondamentale è GRM (*General Resource Modeling*) che fornisce le astrazioni di base per la modellazione di concetti generici e di alto livello per le applicazioni real-time: risorse, servizi, etc...
Ovviamente il livello di granularità necessario per modellare la piattaforma dipende ad esempio dal tipo di piattaforma, dal tipo di applicazioni e dal tipo di analisi che deve essere eseguita sul modello. Gli elementi di questo pacchetto servono quindi per la modellazione di generici sevizi, sia di piattaforme hardware che software, e sono raggruppati in tre pacchetti *ResourceCore*, *ResourceType* e *ResourceManagement*. Il primo definisce gli elementi base e le loro relazioni, il secondo definisce i tipi fondamentali ed i servizi base che essi forniscono, il terzo caratterizza le risorse di gestione ed i suoi servizi associati.
L'allocazione di elementi dell'applicazione su risorse disponibili (ad esempio una piattaforma di esecuzione) è un concetto fondamentale dei sistemi embedded e real time. Il profilo *Allocation* definisce modelli sia di applicazioni rilevanti che di piattaforme di esecuzione. In MARTE un'allocazione è un'associazione tra un'applicazione e una piattaforma di esecuzione (entrambe definite all'interno di MARTE). Gli elementi dell'applicazione possono essere elementi UML che servono per modellare un'applicazione, con aspetti strutturali o comportamentali. Una piattaforma di esecuzione, invece, è rappresentata come un insieme di risorse connesse dove ogni risorsa fornisce servizi per supportare l'esecuzione dell'applicazione. Così le risorse sono elementi di base strutturali, mentre i servizi sono elementi comportamentali.

2.4.3 Sistemi Affidabili (DAM)

Come appena visto, il profilo MARTE ha consentito la specifica di proprietà non funzionali al linguaggio UML. L'introduzione di questo profilo ha fatto fiorire una serie di attività di ricerca incentrate sull'uso e l'arricchimento di tale profilo al fine di estenderne le potenzialità. Uno di questi approcci si è concentrato sulle caratteristiche di dependability possibili in MARTE ma non esplicitamente indirizzate. Questo lavoro ha avuto come capisaldi:

- l'utilizzo di specifiche annotazioni riguardanti aspetti della dependability all'interno di modelli UML;
- la conformità con il profilo MARTE, il quale già prevede un framework per i generici concetti di analisi quantitativa.

Il risultato è quello del profilo Dependability Analysis Modeling (MARTE-DAM) [Bernardi]. Anche questo profilo si struttura in package.

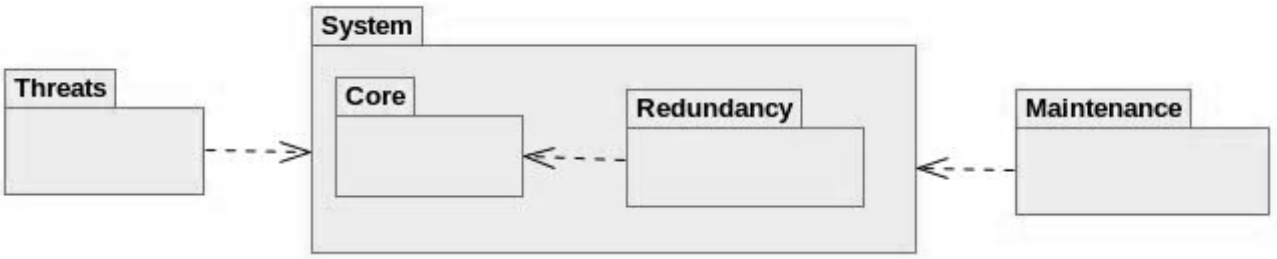

Figura 6: Il profilo MARTE-DAM

Il profilo MARTE-DAM è mostrato il Figura 6. Esso è costituito dai seguenti packages:

System Core (Figura 7): esso fornisce una descrizione del sistema in termini di componenti che, interagendo tra loro eseguendo servizi base, danno vita a servizi di alto livello così come sono percepiti dall'utente che effettua le richieste di servizio. Ogni servizio è implementato da un insieme di step che rappresentano gli stati delle componenti e le azioni.

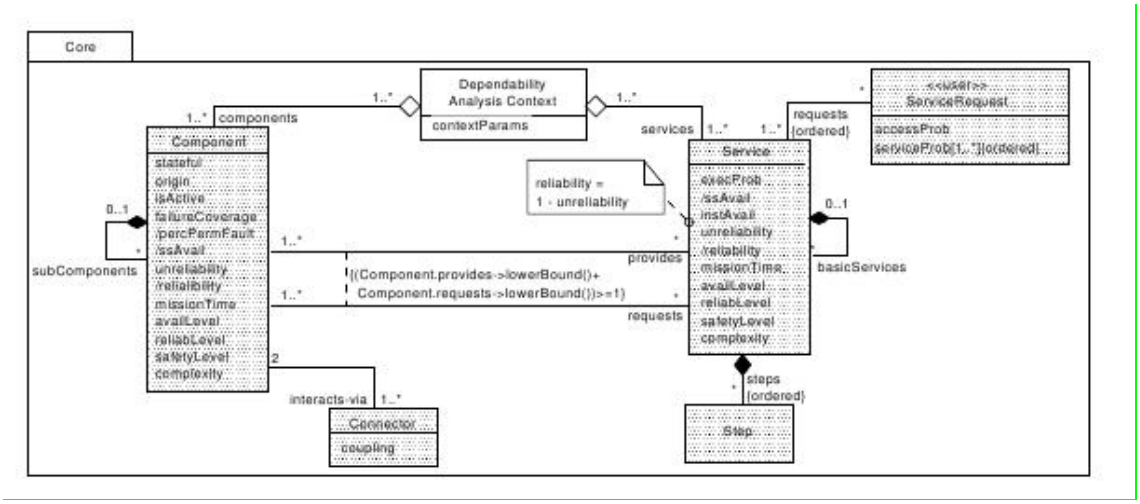

Figura 7: DAM - System Core

Redundancy (Figura 8): fornisce una visione sui concetti di ridondanza e fault tolerance partendo dai concetti di struttura ridondata (Redundant Structure) e successivamente quelli di Spare, Variant, etc...

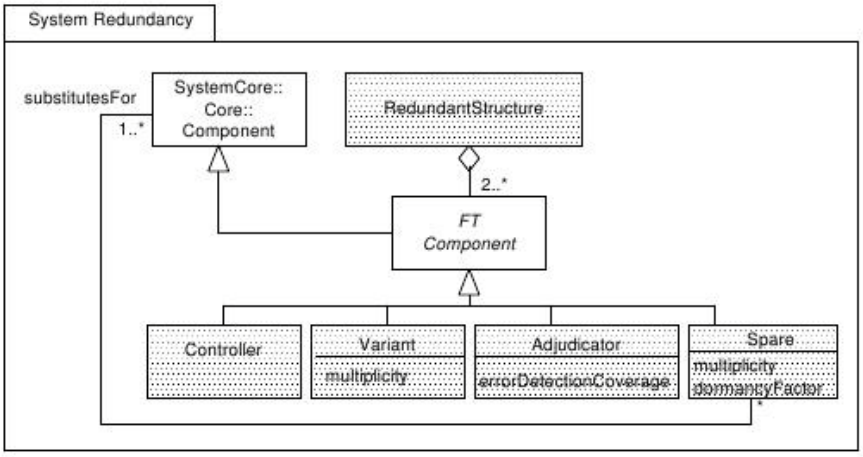

Figura 8: DAM - System Redundancy

Threats (Figura 9): tale modello rappresenta le minacce che possono mettere a rischio il sistema, le relazioni di causa-effetto tra esse ed i collegamenti con le classi del modello *Core*. La classe *impairment* rappresenta un concetto astratto che può essere specializzato in base al tipo del dominio di analisi nel concetto di *failure(malfunzionamento)* o *hazard(pericolo anche per vite umane)* a seconda che si voglia realizzare un'analisi di affidabilità/disponibilità o sicurezza.
La classe *fault* è collegata alla classe *error* da un'associazione causa-effetto così come la classe *error* è collegata alla classe *impairment*. Un *error* può causare diversi tipi di impairment:

- a livello di step di servizio quando il servizio è fornito dal componente ma non in modo corretto
- a livello di componente quando il servizio non viene fornito dal componente
- a livello di sistema quando *l'impairment* è percepito dall'utente.

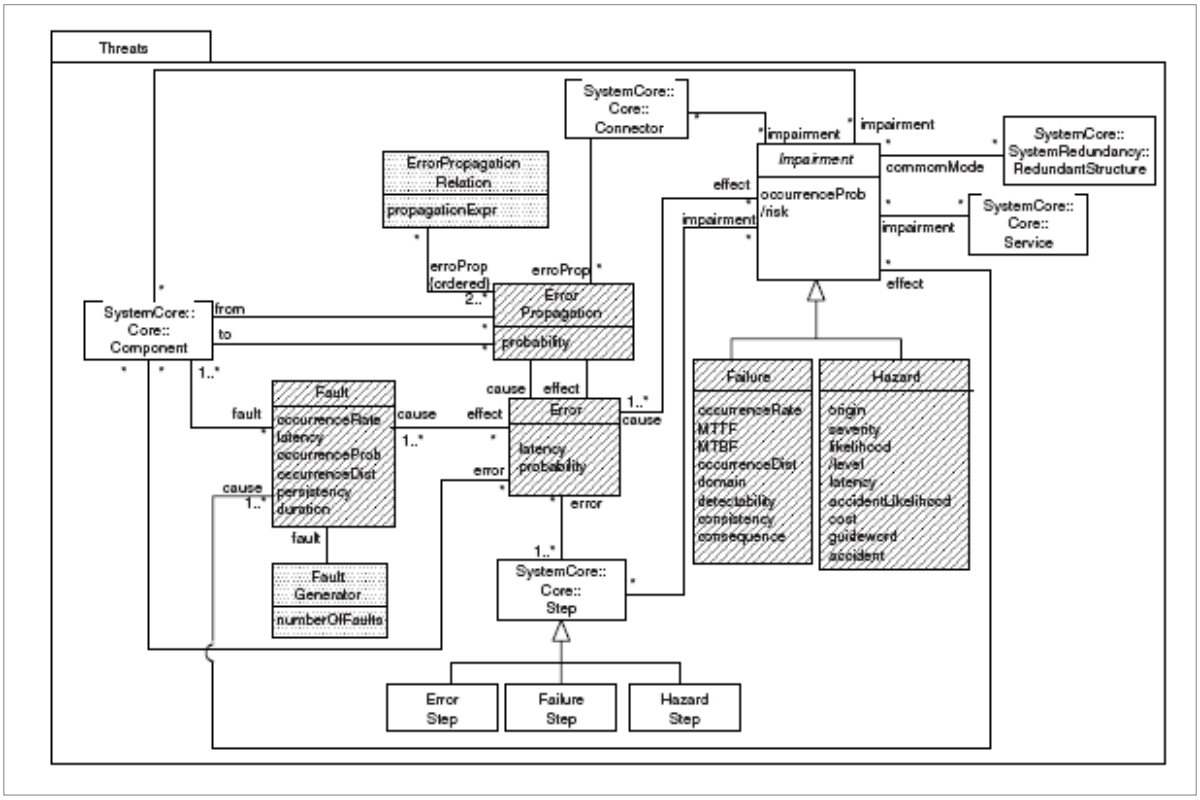

Figura 9: DAM - Threats

Maintenance (Figura 10): tale modello rappresenta le M*aintenanceAction* ossia le azioni da intraprendere per ripristinare lo stato di funzionamento di un sistema su cui si è abbattuto un *threat*. La classe M*aintenanceAction* si specializza in *repairs* delle componenti del sistema che richiedono l'intervento di agenti esterni(operatore umano) o *strategie di recovery* del servizio che, implementate di solito nei sistemi Fault Tolerant si prefiggono l'obiettivo di trasformare lo stato anomalo in uno stato corretto.

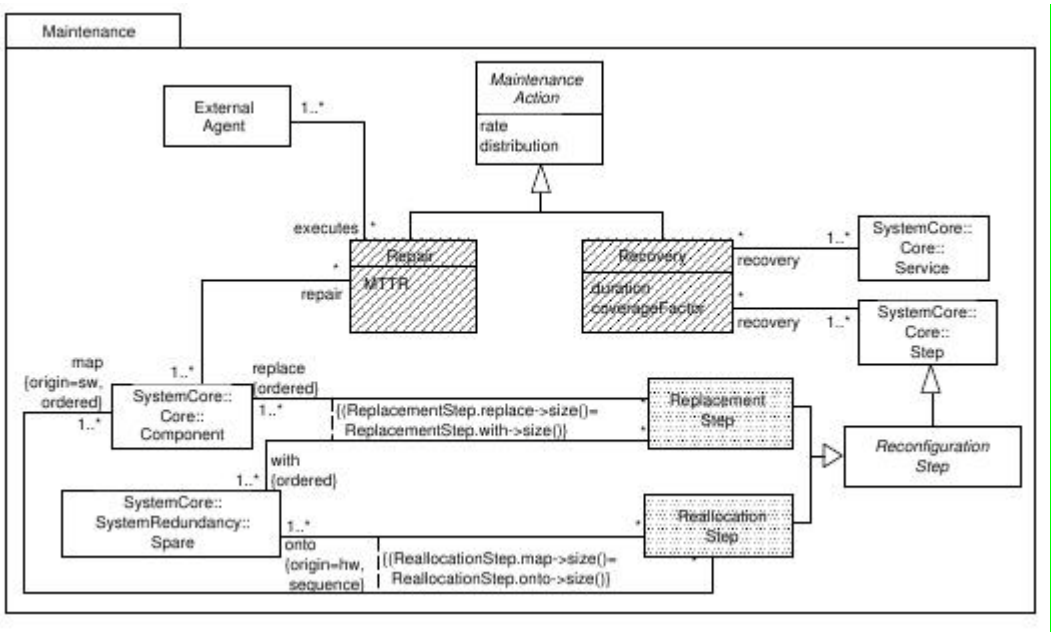

Figura 10: DAM - Maintenance

Capitolo III
Model Driven Formal Modeling

3.1 Approccio metodologico

Descriveremo ora la metodologia definita alla base di questa tesi. L'obiettivo di tale metodologia è fornire uno schema concettuale per la generazione automatica di modelli multiformali a partire da specifiche di alto livello. In particolar modo ci concentreremo sui sistemi critici. La metodologia, come precedentemente illustrato, permette di fondere in un unico approccio i due filoni di ricerca già esistenti nell'ambito della modellazione dei sistemi critici: quello relativo alla generazione automatica di modelli formali attraverso le tecniche ed i metodi di MDE e quello finalizzato alla composizione di modelli multiformali.

Una rappresentazione intuitiva di tale metodologia può essere fornita dallo schema illustrato in Figura 11:

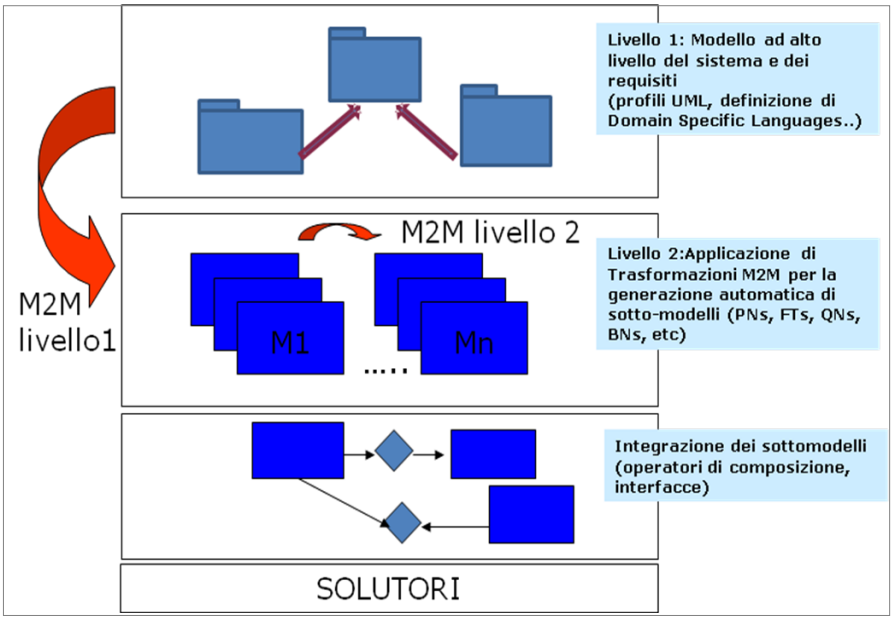

Figura 11: Approccio metodologico

In Figura 11 è possibile individuare i seguenti livelli:

Livello 1- modellazione ad alto livello: questo livello riguarda tutte le fasi di modellazione di alto livello per il sistema ed i suoi requisiti. Come detto in precedenza, al fine di definire una metodologia automatizzabile, è necessario che tutti i modelli siano conformi a linguaggi "formali". A questo livello ci concentremo sui linguaggi atti alla modellazione semplice ed

intuitiva dei sistemi. In particolare possiamo distinguere due fasi: (1) quella di definizione dei linguaggi durante la quale il language engineer definisce il nuovo DSML all'interno del dominio cui si vuole estendere la metodologia e (2) quella dell'uso dei linguaggi. Nella prima fase occorre che il language engineer analizzi con attenzione il dominio in esame e ne partizioni gli aspetti al fine di individuare i vari DSML di interesse (sia orizzontali che verticali che combinazione dei due). Successivamente il language engineer dovrà modellare (in un meta-modello) tali domini e crearne successivamente i linguaggi (attraverso un UML profile o altro). Durante la seconda fase il system modeler utilizzerà quanto definito dal primo utente e creerà un modello del sistema sotto studio, specificando non solo la struttura del sistema stessa ma anche le caratteristiche critiche che il sistema deve possedere (in altre parola modellando anche i requisiti non funzionali).

Livello 2 - modelli multiformali: questo livello contiene i modelli multiformali che catturano, ciascuno con il proprio formalismo, le caratteristiche del sistema in relazione sia alla struttura dello stesso che al tipo di analisi che si vuole effettuare (ossia al tipo di requisito non funzionale che si vuole verificare). Ogni componente o aspetto del sistema ha un suo sottomodello che lo rappresenta e l'intero sistema è visto come unione di tali sottosistemi e dalle relazioni che legano i sottomodelli stessi.

Livello 3- composizione: a questo livello un nuovo modello multiformale è presente; in esso, le relazioni diventano veri e propri operatori di composizionalità che definiscono le regole di passaggio dei parametri e/o di risultati tra un modello e l'altro abilitando non solo una descrizione (multi-)formale del sistema ma anche e soprattutto un processo risolutivo del modello in grado di analizzare ogni singolo sottomodello con il solver più adatto e di comporre i risultati nel modo più adeguato. Questo livello non è stato trattato approfonditamente all'interno di questa tesi. Si rimanda ai seguenti riferimenti [Vittorini] per ulteriori approfondimenti.

In tale schema rivendicano una loro identità le seguenti trasformazioni:

M2M di livello 1: queste trasformazioni tra modelli permettono di generare modelli multiformali a partire dalle specifiche di alto livello;

M2M di livello 2: sfruttano le equivalenze e le semantiche traslazionali possedute da uno o più linguaggi formali per permettere un passaggio dall'uno all'altro nell'ottica di semplificare il modello multiformale e di conseguenza il processo risolutivo (es. i fault trees possono essere espressi attraverso una Generalized Stochastic Petri Nets ed è quindi possibile creare una trasformazione da FT a GSPN).

3.2 Tecniche abilitanti

Al fine di poter rendere possibile e scalabile la metodologia esposta, è necessario uno studio più approfondito su due punti: quello della definizione di nuovi linguaggi e quello sulla definizione di trasformazioni tra modelli. In questa tesi verrà posto uno stress particolare sul problema della "scalabilità" della metodologia esposta, ossia la capacità della metodologia stessa ad essere applicata in situazioni reali ed in ambiti industriali. Una delle più grandi illusioni possibili nell'ambito del Model Driven Engineering è infatti quello di pensare che la sola applicazione di tali tecniche consente l'ottenimento del desiderato "silver bullet" dell'ingegneria del software e dei sistemi. In realtà la complessità delle applicazioni viene spostata ora sui processi di definizione di linguaggi e tecniche. L'unico vero silver bullet resta quello del riuso ed è obbligatorio dunque ricercare nuove tecniche che favoriscano il riuso degli artefatti esistenti sia da una parte che dall'altra.

Nel Capitolo successivo verranno, dunque, studiate tecniche per la derivazione di linguaggi ed introdotte tecniche di ereditarietà tra trasformazioni. Al fine di limitare il campo di indagine della ricerca ci focalizzeremo sul problema della riparabilità.

3.3 Architettura di riferimento

In questa sezione cercheremo di individuare gli elementi prevalenti di un'architettura realizzativa a supporto della metodologia espressa. L'architettura ha il ruolo di definire il grado di automatizzazione della metodologia e gli strumenti in grado di realizzarla. L'architettura è raffigurata in Figura 12.

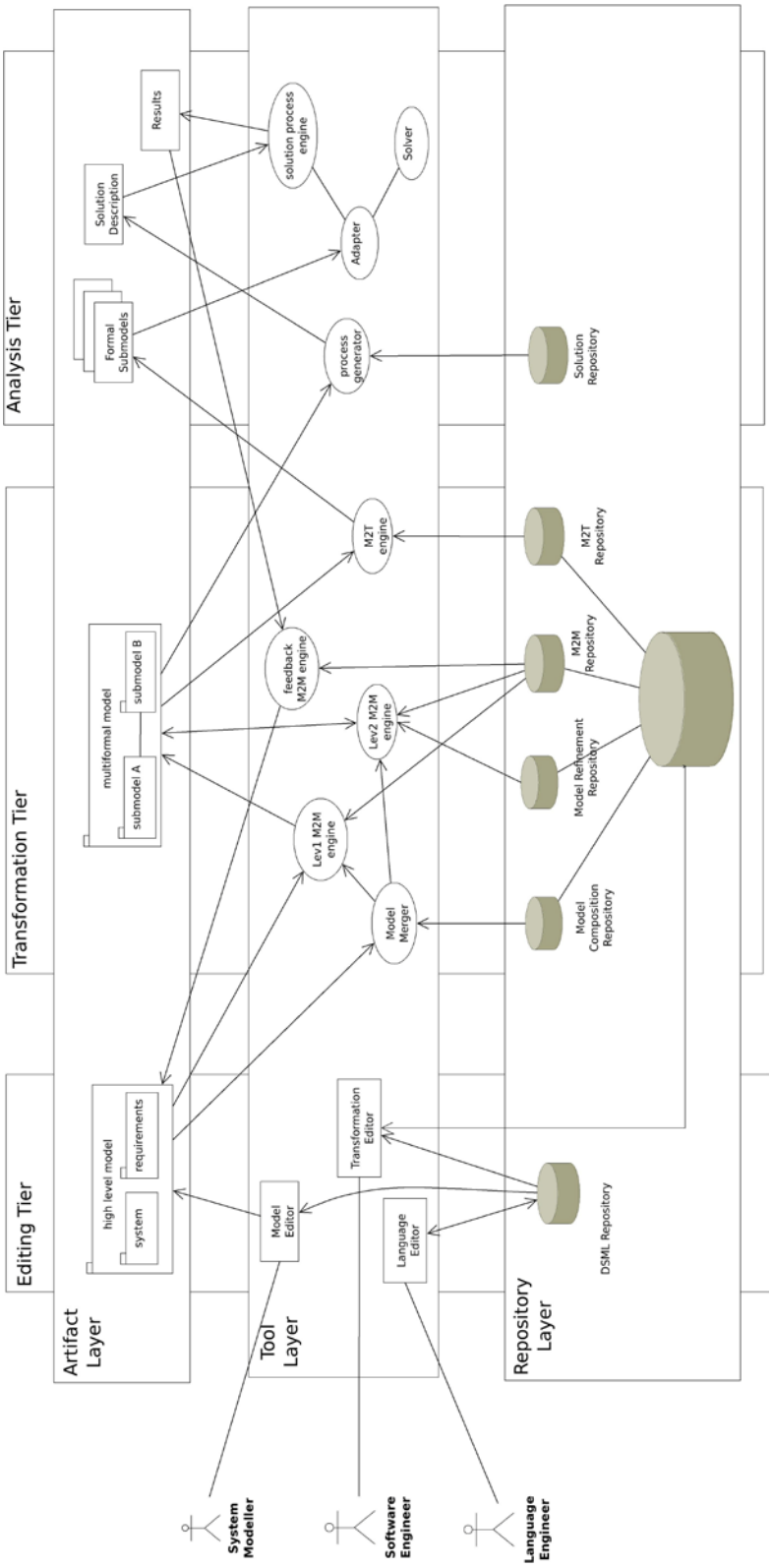

Figura 12: Architettura di riferimento

Questa architettura è caratterizzabile come three-layer, three-tier e three-user. Ci sono infatti 3 tier verticali (Editing tier, Transformation tier e Analysis tier) che rappresentano le tre macrofasi del processo proposto; ci sono inoltre 3 layer orizzontali (Artifact Layer, Tool Layer e Repository Layer) che rappresentano gli oggetti (modelli, strumenti e data base) coinvolti in ogni tier.

A partire dall'User Tier, descriviamo ogni elemento architetturale cercando di evidenziarne le caratteristiche principali. Si noti che questa architettura non è stata implementata interamente ma che comunque diversi strumenti commerciali ed open source disponibili possono essere usati all'interno di essa. Notiamo inoltre che il tier finale dell'architettura ricalca in parte OMF, l'OsMoSys Solution Framework già sviluppato all'interno del progetto OsMoSys dalla Università degli Studi “Federico II” e dalla Seconda Università di Napoli [DiLorenzo].

Lo user tier comprende:

- *High level model*: modello di alto livello coincidente con l'artefatto del livello 1 della metodologia;
- *Model Editor*: semplice editor di modelli;
- *Language Editor*: semplice editor di linguaggi;
- *Transformation Editor*: semplice editor di trasformazioni tra modelli (M2M e M2T);
- *DSML Repository*: data base che contiene i DSML creati e le relazioni (vedi Capitolo successivo) tra i linguaggi stessi.

Il transformation tier comprende:

- *multi-formal model*: modello multiformale corrispondente agli artefatti relativi al livello 2 della metodologia proposta;
- *lev 1 M2M engine*: motore di model transformation di livello 1;
- *lev 2 M2M engine*: motore di model transformation di livello 2 (i due motori che sono raffigurati distinti nel disegno architettuale possono essere “incarnati” nello stesso componente fisico);
- *model composer*: questo modulo analizza il modello di alto livello ed indica agli engine di model transformation come comporre i modelli multiformali: indica agli engine di model transformation quali operatori di composizionalità inserire nel modello formale;

- *M2T engine*: motore trasformazionale in grado di generare testo a partire dai modelli multiformali;
- *feedback M2M engine*: motore che traduce i risultati dell'analisi a basso livello nel modello di alto livello al fine di dare un corretto feedback all'utente;
- *transformation repository*: al livello dei repository, esistono per questo tier diversi database inerenti ai diversi aspetti da memorizzare per il funzionamento degli engine; tutti questi database sono incorporati in un unico repository.

L'Analysis Tier comprende:

- *i sottomodelli formali*: descritti nei linguaggi atti ad essere analizzati dai solver;
- *Process Generator*: motore che trasforma il modello multiformale in ingresso generando un processo di soluzione (ossia una sequenza di operazioni di invocazione di adapter su sottomodelli specifici e di trasformazione di parametri e risultati dei modelli stessi);
- *Solution Process Engine*: descrizione del processo di invocazione degli adapter e di risoluzione dei singoli modelli nell'ambito della risoluzione del modello multiformale;
- *Result*: risultati dell'analisi del processo di soluzione del modello multiformale;
- *Solver*: strumento esterno specializzato nell'analisi di un formalismo specifico;
- *Adapter*: strumento che incapsula un solver al fine di fornire all'esterno (ossia al Solution Process Engine) un'interfaccia unificata;
- *Solution Repository*: archivio per la memorizzazione delle traduzioni dei processi risolutivi multiformali in serie di invocazioni di adapter specifici per compiere il compito.

3.4 Strumenti a supporto

L'architettura introdotta non deve necessariamente essere implementata da zero in quanto le tecniche del Model Driven Engineering sono ampiamente supportate non solo dalla letteratura scientifica ma anche da un crescente interesse industriale, alimentato questo da un solido supporto da parte di strumenti esistenti ed ormai in via di consolidamento.

Sono stati dunque usati a tale scopo diversi strumenti per:

- modellazione e meta-modellazione: il tool Papyrus e la piattaforma EMF (Eclipse Modeling Framework) permettono facilmente sia di meta-modellare che di meta-meta-

modellare nell'ambito di diversi stack di linguaggi; Papyrus permette di lavorare con modelli UML, SYSML sia nella definizione di modelli che in quella di profili. Diversi profili standard (tra i quali anche MARTE) sono ormai disponibili per questo tool che è conforme anche agli standard per l'interoperabilità (XMI, Diagram Interchange Format, etc..). EMF permette invece di creare linguaggi conformi al meta-meta-modello ECORE;

- model transformation: sono stati considerati essenzialmente due linguaggi per la model transformation: lo standard OMG QVT (Query View Transformation) ed il linguaggio ATL (Atlas Transformation Language) [Jouault]. Il primo prevede tre diversi livelli di linguaggio chiamati Relations, Core ed Operational Mappings. I primi due rappresentano la parte dichiarativa della metodologia mentre Opertional Mappings arricchisce il potere espressivo del linguaggio con costrutti imperativi. Il secondo permette attraverso le diverse regole esprimibili in esso, sia costrutti imperativi che dichiarativi. Sia ATL che QVT risultano ben supportati da strumenti (specialmente ATL con ATL Eclipse, strumento ormai ben maturo ed in grado anche di fornire un ambiente di debugging delle trasformazioni create).

Comune denominatore di tutti questi strumenti è la piattaforma di programmazione/modellazione Eclipse che si sta affermando sempre più come strumento unificante per la modellazione e la meta-modellazione nonché per la generazione automatica di modelli e codice.

Capitolo IV
Il riuso nei linguaggi e nelle trasformazioni

4.1 L'ereditarietà nei linguaggi

In questo capitolo vengono descritti i principi e le tecniche utilizzate per la creazione di nuovi linguaggi per la specifica di sistemi e proprietà e per l'analisi. Il problema principale è cercare tecniche per definire nuovi linguaggi in modo scalabile ed incrementale. Gli argomenti trattati sono stati oggetto di pubblicazione in [Bernardi2] ed in [Marrone].

Un approccio risolutivo a tale problema è costituito dalla possibilità di definire gerarchie di linguaggi basate sulla relazione di ereditarietà. Un esempio di tale relazione è raffigurato in Figura 13.

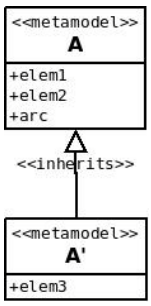

Figura 13: Gerarchie di meta-modelli

In questa figura è schematizzato il meccanismo con cui modellare il fatto che il linguaggio A′ "aggiunge" al formalismo A l'elemento *elem3* venendo così ad essere costituito da tutti gli elementi linguistici di A e da quest'ultimo elemento.

Un esempio reale di tale meccanismo è costituito dal linguaggio dei Repairable Fault Trees: questo linguaggio eredita da quello dei Fault Trees. In Figura 14 ed in Figura 15 sono raffigurati rispettivamente i meta-modelli del secondo e del primo linguaggio.

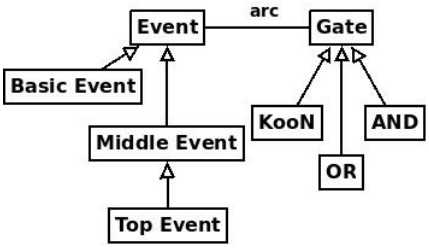

Figura 14: Meta-modello Fault Tree

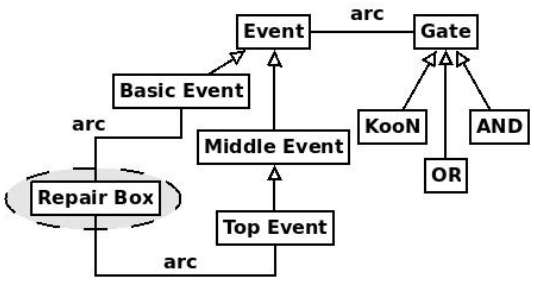

Figura 15: Meta-modello Repairable Fault Tree

Da tali figure si evidenzia come il linguaggio degli RFT aggiunga al linguaggio dei Fault Trees il concetto di *Repair Box* ed utilizzi l'*arc* dei Fault Trees per connettere questo nuovo elemento ad altri elementi di FT (*Top Event* e *Basic Event*).

Partendo dall'esempio specifico del linguaggio RFT vogliamo definire una classificazione dei modi di riuso di un elemento linguistico all'interno di un formalismo derivato (vedi Figura 16).

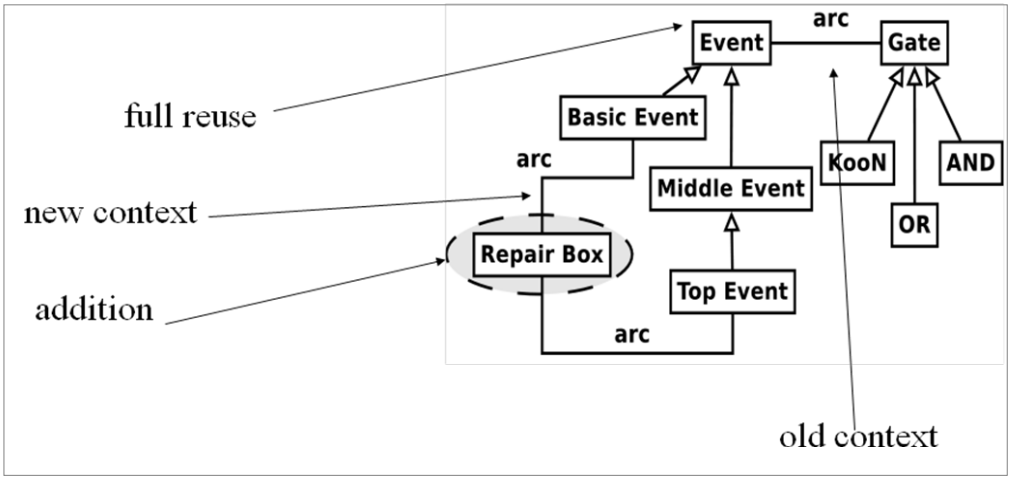

Figura 16: Riuso degli elementi in un formalismo

Sono state individuate quattro classi di riuso che possono essere estese ovviamente ad un generico formalismo:

- *addition*: il formalismo derivato ha un nuovo elemento oltre agli elementi ereditati;
- *ridefinition*:
  - *new context*: un elemento presente nel formalismo padre può essere accoppiato a nuovi elementi presenti nel formalismo figlio;

- *old context*: un elemento presente nel formalismo padre assume un nuovo significato nel formalismo figlio anche se il contesto in cui viene usato non cambia;
- *full reuse*: un elemento definito nel formalismo padre è ereditato senza alcuna modifica.

Questo meccanismo può essere esteso anche a linguaggi UML-based (i profili) dove un nuovo linguaggio può essere generato da uno precedente aggiungendo un nuovo package e definendo le opportune relazioni di dipendenza con il package precedente. Se consideriamo, ad esempio, la struttura di MARTE-DAM in Figura 6, possiamo immaginare una situazione in cui l'intero linguaggio MARTE-DAM non sia altro che un linguaggio che eredita dal sottolinguaggio costituito dai package *Core* e *Threats* con l'aggiunta degli elementi costituenti il package *Maintenance* e delle dipendenze tra questo ed i primi package.

L'uso di questo formalismo nella definizione di strutture di linguaggi e la possibilità di ereditarietà multipla permettono la creazione di gerarchie di linguaggi (sia di alto che di basso livello di astrazione) molto complesse ed espressive. Questi meccanismi sono alla base della definizione di DSML sofisticati, ottenuti attraverso la composizione di DSML orizzontali e verticali nonché di gerarchie di linguaggi formali complessi come quello rappresentato in Figura 17.

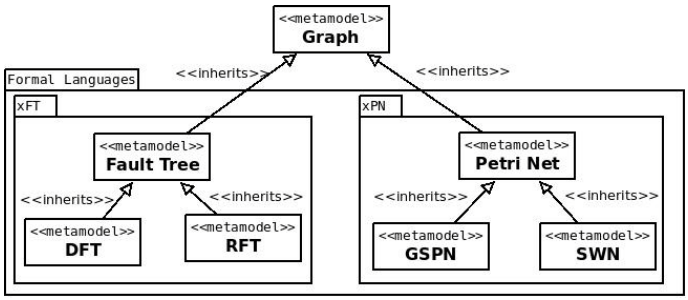

Figura 17: Una gerarchia di linguaggi formali

4.1.1 Estensione di MARTE-DAM

Nell'ambito dell'estensione di linguaggi per sistemi riparabili, un discorso a parte deve essere dedicato ai linguaggi di alto livello. Un primo passo verso l'obiettivo finale è quello di proporre di aumentare le capacità di modellazione ed analisi del profilo UML MARTE-DAM relativamente agli aspetti di ridondanza e manutenzione.

Per quanto riguarda gli aspetti di fault tolerance un'analisi del profilo MARTE-DAM ha evidenziato alcuni margini di miglioramento in tale package. L'estensione proposta, raffigurata in Figura 18, consiste nell'aggiunta del tagged value *FTlevel* allo stereotipo *Redundant Structure*. Il valore di questo tagged value indica quante repliche dei componenti o degli spare contenuti nel package devono essere attivi contemporaneamente per

conservare la fault tolerance. Operando opportunamente su tale valore possiamo ottenere diversi livelli di fault tolerance (1-out-of-N, N-out-of-N, k-out-of-N).

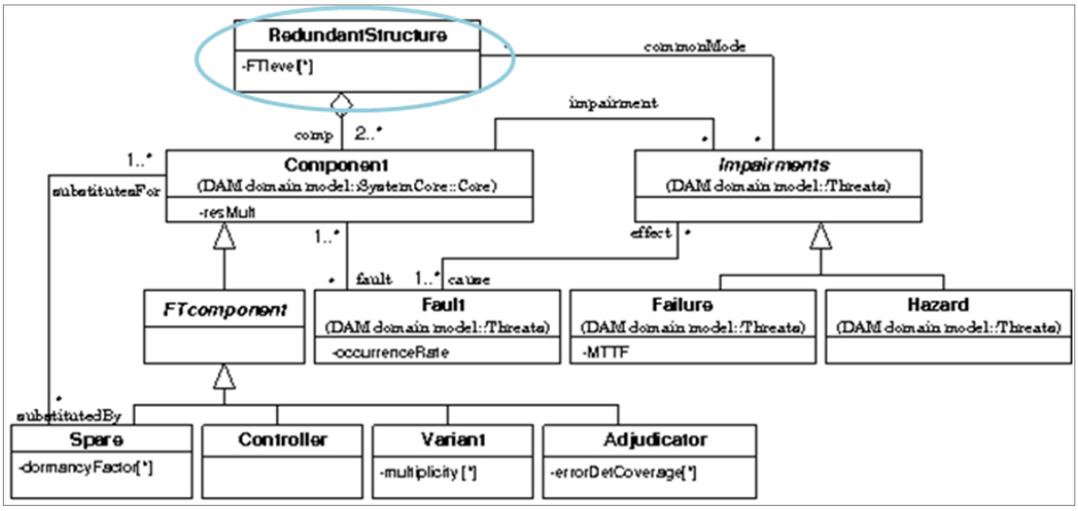

Figura 18: Estensione del package Redundancy

La seconda più consistente estensione del profilo MARTE-DAM è costituita dagli aspetti legati alla manutenzione ed alla riparabilità. In Figura 19 è raffigurato il domain model relativo al package maintenance di MARTE-DAM esteso in alcuni punti:

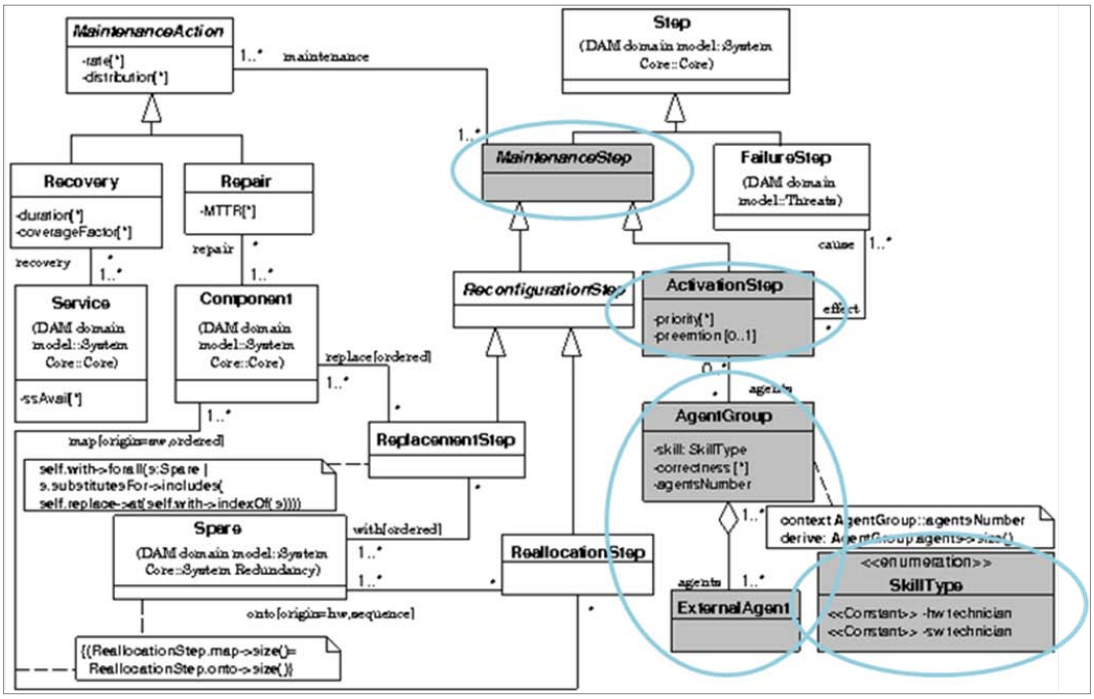

Figura 19: Estensione del package Maintenance

- introduzione dell'astrazione *Maintenance Step*: uno step è una qualsiasi azione atomica che costituisca un'azione complessa all'interno di MARTE. Un Maintenanace Step è un concetto base per la definizione successiva di eventi concernenti la manutenzione/riparazione di un sistema;
- *Activation Step*: specializzazione del Maintenance Step, è relativo agli eventi che scatenano un'azione di Maintenance (fallimenti, diagnostiche, procedure di manutenzione preventiva, etc...);
- *Agent Group*: introduzione del concetto di squadra di manutenzione spesso necessaria per svolgere compiti di sostituzione e/o diagnosi complesse;
- *SkillType*: differenziazione delle caratteristiche di abilità degli agenti, necessari per costituire squadre eterogenee/omogenee a seconda dei casi.

4.2 Ereditarietà di trasformazioni

Altro importante contributo originale della tesi è costituito dalla definizione di una tecnica per la costruzione di trasformazioni tra modelli complesse che, sfruttando l'ereditarietà presente nei linguaggi, induca un altrettanto importante relazione di ereditarietà nelle trasformazioni. Gli argomenti trattati sono stati oggetto di pubblicazione in [Bernardi2] ed in [Marrone].

L'idea di base è schematizzata in Figura 20.

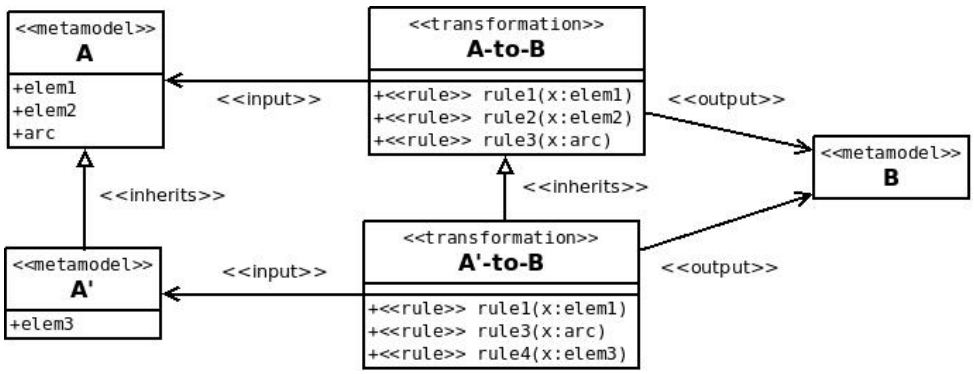

Figura 20: Ereditarietà tra trasformazioni - schema di principio

Possiamo formulare il problema come segue: date due trasformazioni tra modelli, la prima dal linguaggio *A* a quello *B* (*A-to-B*) e l'altra dal linguaggio *A′* a quello *B* (*A′-to-B*), dove

esista una relazione di ereditarietà tra i linguaggi *A* ed *A′*, ci chiediamo in che relazione esistano le due trasformazioni sopradette.

In base alla classificazione introdotta precedentemente, si farà riferimento ai linguaggi di trasformazioni basati su regole (*rule*). In tale contesto una trasformazione sarà caratterizzata da un insieme di rules attivate da elementi che appartengono ad un linguaggio sorgente e che producono elementi in un linguaggio destinazione. Supponiamo che la trasformazione *A-to-B* sia stata sviluppata da zero e che sia disponibile per il riuso. Analizziamo adesso come la classificazione dei modi di riuso introdotta al paragrafo precedente induce un comportamento, una sorta di ereditarietà tra trasformazioni M2M favorendo il riuso della trasformazione M2M ereditata e definendo meccanismi per l'acquisizione e la modifica di regole esistenti o l'aggiunta di nuove regole all'interno della trasformazione *A′-to-B*:

- *addition*: in questo caso deve essere definita una nuova regola che trasforma l'elemento del meta-modello sorgente *A′* in un elemento di quello destinazione. La Figura 20 A′ contiene il nuovo elemento *elem3*: dovrà, dunque, essere definita la trasformazione che tramuta *elem3* in un elemento del meta-modello destinazione *B* (è il caso della *rule4(x:elem3)*);

- *ridefinition*:

  - *new context*: l'insieme di rule della trasformazione *A′-to-B* può essere esteso e/o parzialmente ridefinito in *A′-to-B*. E' questo il caso dell'elemento *arc* e della *rule3(x:arc)* che deve essere ridefinito in modo tale che essa scatta se un *arc* connette *elem1* ad *elem3*: nella trasformazione la regola *rule3* di *A′-to-B* sovrascrive la stessa relativa già presente in *A-to-B*.

  - *old context*: le regole di *A-to-B* relative a questi elementi devono essere ridefiniti in *A′-to-B* al seguito del cambiamento di significato che questi elementi assumono nel nuovo formalismo (questo è il caso di *rule1(x : elem1)* che è ridefinita in *A′-to-B*);

- *full reuse*: le regole della trasformazione "base" vengono riusate senza alcuna modifica (è questo il caso della regola *rule2(x : elem2)*).

Questa metodologia di derivazione è supportata da due meccanismi abilitanti: la superimposition e la rule overriding. Il primo consente l'esecuzione di diverse trasformazioni sullo stesso modello generando virtualmente una trasformazione costituita dall'unione delle rule di tutte le trasformazioni coinvolte. In pratica rende possibile che una trasformazione erediti tutte le rule di una trasformazione superimposta. D'altra parte la rule overriding consente di ridefinire una rule già definita nella trasformazione superimposta. I

principali linguaggi di definizione delle trasformazioni (come ad esempio ATL) implementano questi meccanismi.

4.3 Una transformation chain per sistemi riparabili

Le tecniche sopra esposte sono state applicate alla definizione di una catena di trasformazioni nell'ambito della modellazione ed analisi di sistemi riparabili. Dal punto di vista dei linguaggi, sono stati definiti linguaggi di alto livello (MARTE-DAM), linguaggi multiformali (RFT). L'analisi di modelli conformi a questo ultimo linguaggio è però possibile a patto di tradurre il modello stesso in una Petri Nets. In Figura 21 è illustrato il meta-modello relativo al linguaggio delle Generalized Stochastic Petri Nets (GSPN).

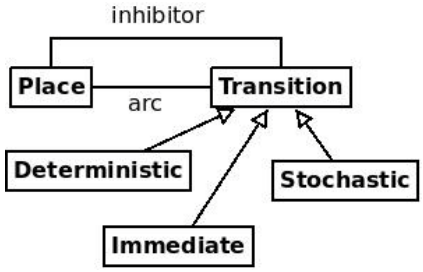

Figura 21: Meta-modello del formalismo GSPN

In Figura 22 è illustrata la catena delle trasformazioni (di livello 1 e di livello 2 in accordo con la metodologia definita nel capitolo precedente).

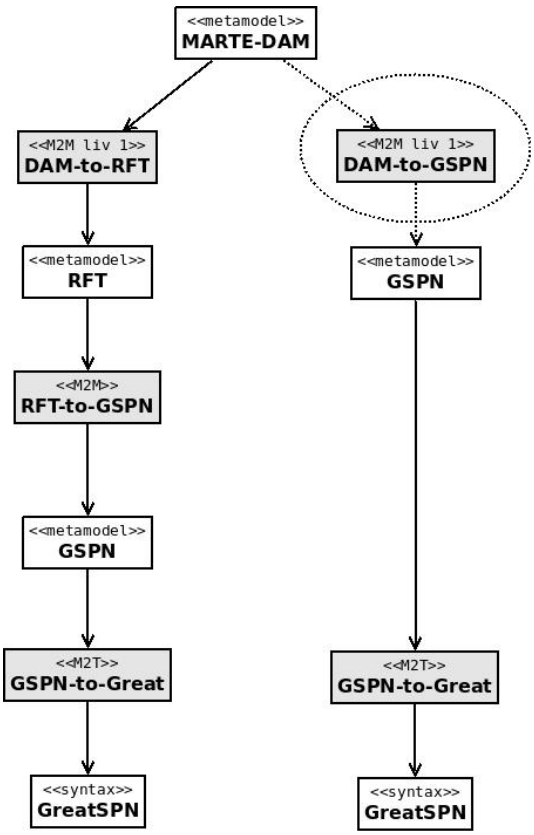

Figura 22: Catena di trasformazioni per la riparabilità

Questa catena di trasformazioni è composta da quattro trasformazioni (rappresentate con box grigi):

- *DAM-to-RFT*: questa trasformazione si pone al livello 1 della nostra metodologia e genera, a partire da un modello conforme al profilo MARTE-DAM, un modello RFT a meno della politica di riparazione che, in accordo con quanto descritto sul formalismo RFT, è espressa attraverso una GSPN;

- *DAM-to-GSPN*: questa trasformazione si pone al livello 1 e si concentra, a partire dal profilo MARTE-DAM, sull'analisi della politica di riparazione e sulla generazione della Petri Net che la implementa;

- *RFT-to-GSPN*: questa trasformazione si pone al livello 2 e ha l'obiettivo di tradurre la struttura di un modello RFT (a meno della politica di riparazione) in una GSPN;

- *GSPN-to-Great*: l'ultima trasformazione (l'unica di tipo model-to-text) è necessaria a tradurre il modello GSPN in una sintassi concreta comprensibile da un solver specifico nell'analisi.

Al fine di dimostrare la fattibilità della metodologia definita, sono state implementate tutte le trasformazioni eccetto *DAM-to-GSPN* (la trasformazione nell'ovale tratteggiato)

riservando il completamento della catena di trasformazioni a sviluppi futuri. Nel seguito daremo ulteriori dettagli sulla progettazione delle trasformazioni *DAM-to-RFT* e *RFT-to-GSPN* e daremo invece dettagli tecnici e di implementazione di tutte le trasformazioni implementate.

4.3.1 La trasformazione DAM-TO-RFT

E' possibile, sulla base della strutturazione gerarchica dei linguaggi MARTE-DAM e RFT e secondo il paradigma basato sull'ereditarietà dei linguaggi prima espressa, pensare di progettare la trasformazione DAM-TO-RFT in modo incrementale. La Figura 23 esprime quanto fatto; la trasformazione infatti è ottenibile come composizione di due trasformazioni:

- *dam2ft*: questo modulo traduce il sottoinsieme dei concetti di DAM relativi alla struttura, alla fault tolerance ed ai guasti in un fault tree;
- *dam2rft*: questo modulo traduce la parte di riparazione di DAM aggiungendo al Fault Tree precedentemente generato i concetti presenti nel linguaggio RFT e non presenti in FT.

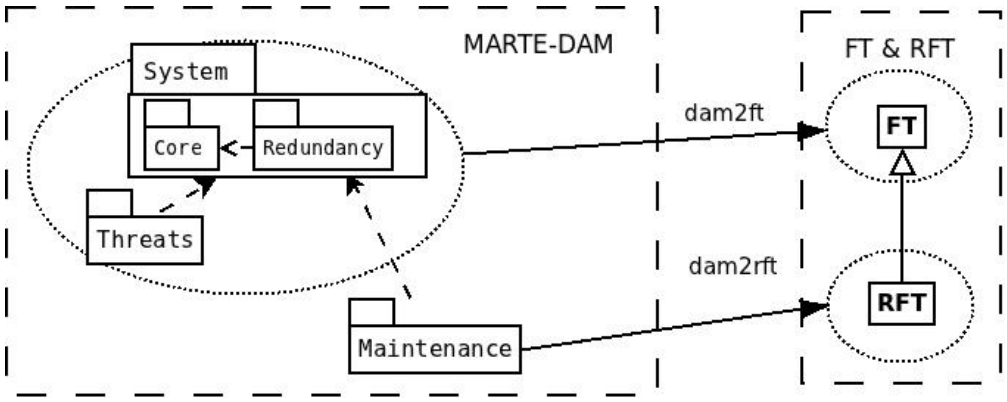

Figura 23: DAM-to-RFT

In particolare, avendo notato che il profilo MARTE-DAM eredita dal sottoinsieme del profilo stesso legato ai package *System* e *Threats*, ed essendo il linguaggio RFT derivato da quello FT, possiamo definire la trasformazione globale come risultato della superimposizione della trasformazione *dam2rft* su *dam2ft*.

Analizziamo ora le singole trasformazioni.

La trasformazione ***dam2ft*** agisce sugli stereotipi contenuti nei package *System* e *Threats*: in particolare i concetti chiave necessari alla nostra trasformazione sono i seguenti stereotipi del profilo MARTE-DAM: *DaComponent* (che caratterizza un Component di UML aggiungendogli la capacità di modellare le caratteristiche di dependability di componenti hardware del sistema), *DaRedundantStructure* (che, applicato a package di UML, permette

di definire dei “contenitori” di componenti in grado di garantire tra essi una ridondanza secondo le caratteristiche descritte precedentemente) e *DaService* (che indica quale use case è etichettato come “funzionalità critica” e quindi soggetto ad analisi di manutenzione). Per illustrare il funzionamento della trasformazione supporremo dunque di possedere, nel modello sorgente, una “vista“strutturale” (legata all’organizzazione del sistema in *DaComponent*) e una “vista funzionale” (legata invece alla definizione delle funzionalità esterne, agli use case taggati con DaService).

In Figura 24 sono rappresentate le regole di traduzione dal (sotto-)modello MARTE-DAM al (sotto-)modello RFT.

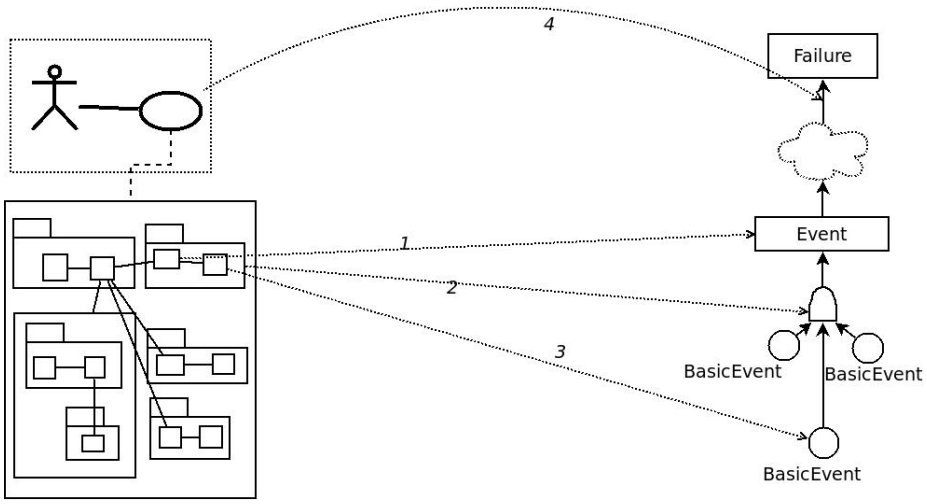

Figura 24: dam2ft: schema di funzionamento

Possiamo sintetizzare la trasformazione nel seguente modo:

1. si parte dal DaComponent più esterno che viene trasformato in un middle event (Event);

2. si analizza il package che contiene tale *DaComponent*, se stereotipato con il tag *DaRedundantStructure*, si analizza il valore del *FTlevel* in relazione anche al valore della somma N del numero delle repliche di *DaComponent* e *DaSpare* contenuti all’interno del package (ossia la somma dei valori dei tagged value *resMult* degli stereotipi *DaComponent* e *DaSpare*):

    a. *FTlevel* pari ad 1: in questo caso, basta un solo *DaComponent* attivo per rendere attiva la struttura ridondata, cioè è necessario che tutti gli eventi di livello inferiore nel FT siano in errore (logica relativa al gate AND) affinché il guasto di propaghi;

b. *FTlevel* pari ad N: in questo caso, è necessario che tutti i *DaComponent* contenuti siano non in errore per tenere attiva la struttura ridondata, cioè basta che un solo evento di livello inferiore nel FT sia in errore (logica relativa al gate OR) affinché il guasto di propaghi;

c. *FTlevel* pari a K (minore di N): sulla base di quanto detto, il costrutto ridondato corrisponde ad un gate (N-K)-out-of-N;

3. si itera ricorsivamente il procedimento sui *DaComponent* contenuti nel package; il procedimento termina quando si trova un *DaComponent* non scomponibile e caratterizzato da un tasso di guasto non nullo (tagged value *MTTF* dello stereotipo *DaComponent*): in questo caso viene generato sul fault tree un Basic Event.

4. Viene cercato un *DaService* nel modello e si cerca il tagged value *usedResource* che indica il *DaComponent* che implementa il servizio; sulla base di questa indicazione, si genera un top event del fault tree che viene collegato al middle event generato dal *DaComponent* nel processo precedente.

Di seguito viene riportata la called rule che implementa il processo ricorsivo descritto dai passaggi 1-3.

```
rule componentTranslation(cx: UML!Component) {
  do  {
    -- generation of middle event
    thisModule.generateME(CX.name);
    -- generation of gate block
    if (cx.owner.isRedundant()) {
      if (cx.owner.getFTlevel() == 1){
        -- AND gate
        thisModule.genAND(cx.name + '_gate');
      } else if (cx.owner.getFTlevel() == cx.owner.getTotalComponent()) {
        -- OR gate
        thisModule.genOR(cx.name + '_gate');
      } else {
        -- KooN gate
        n<- cx.owner.getTotalComponent();
```

```
        thisModule.genKooN(cx.name + '_gate', n-cx.owner.getFTlevel());
      }
      -- adding gate outgoing arc
      thisModule.genGateToEvent(cx.name + '_gate',cx.name);
      -- Basic events generation
      if (cx.isFaulty()) {
        for(num in thisModule.seq->select(s|s<=cx.owner.getTotComp())) {
          thisModule.genBasicEvent(cx.name + num,cx.getMTBF());
          thisModule.genEventToGate(cx.name + num,cx.name + '_gate');
        }
      }
    }
  }
}
```

Si noti l'utilizzo di diversi helper e called rules all'interno del codice. L'opportunità di usare questi helper è ovviamente legata alle regole di buona programmazione. Nella fattispecie:

- *generateME(...)*: called rule che genera materialmente nel modello di uscita un MiddleEvent relativo al metamodello FT;
- *isRedundant()*: helper che ritorna true se il package su cui è invocato è stereotipato come DaRedundantStructure;
- *getFTlevel()*: helper che ritorna il valore del tagged value FTlevel in relazione al DaRedundantStrucutre;
- *genAND(…)*: called rule che genera una porta AND nel Fault Tree;
- *genKooN(…)*: called rule che genera una porta KooN nel Fault Tree;
- *genOR(…)*: called rule che genera una porta OR nel Fault Tree;
- *genGateToEvent(…)*: called rule che genera un arco di collegamento tra un gate ed un middle event;
- *isFaulty()*: helper che indica se il DaComponent possiede un valore (non nullo) relativamente al tagged value MTTF;

- *getTotComp()*: helper che calcola il numero di DaComponent e di DaSpare presenti nella DaRedundantStructure;
- *genBasicEvent()*: called rule che genera un Basic Event;
- *getMTBF()*: helper che ritorna il valore del tagged value MTTF;
- *genEventToGate()*: called rule che genera un arco di collegamento tra un evento ed un gate.

La trasformazione ***dam2rft*** agisce, invece, sulla parte di riparazione andando ad aggiungere RepairBox e gli archi che li collegano agli eventi del FaultTree. Nella fattispecie utilizzeremo i concetti presenti nel package Maintenance del modello di dominio di MARTE-DAM e gli stereotipi corrispondenti: *DaComponent* (in quanto detentore delle informazioni legate alla riparabilità di un componente (gli MTTR) e *DaActivationStep*. In Figura 25 è evidenziata la traduzione del modello MARTE-DAM in un RFT: sin noti che le parti precedentemente tradotte sono state tracciate in grigio.

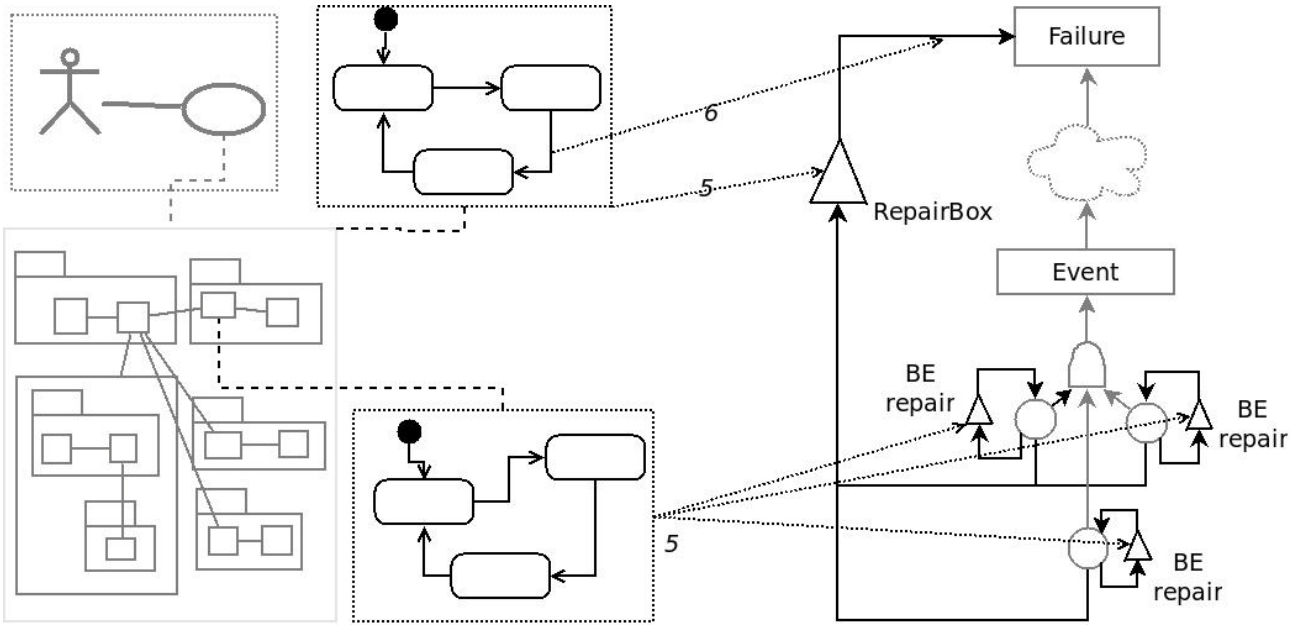

Figura 25: dam2rft: schema di funzionamento

Possiamo sintetizzare la trasformazione nel seguente modo:

5. la presenza di un modello di riparazione scatena di per sè la generazione di un RepairBox nel RFT;

6. si analizza il modello e si individua il *DaActivationStep*. Il *DaComponent* indicato nel tagged value *cause* individua il *DaComponent* che scatena la riparazione e quindi (passando al modello RFT di destinazione) l'evento che deve essere collegato alla Repair Box attraverso un triggering arc. A partire da questo evento si ridiscenda l'albero in modo ricorsivo fino ad individuare i Basic Event che dovranno essere collegati alla Repair Box come repair arc.

Di seguito la matched rule che implementa il punto 5.

```
rule repairBox {
      from
            r: UML!Component (r.isRepairable())
      to
            rb: RFT!RepairBox (name<-r.name,rate<-r.getMTTR())
}
```

4.3.2 La Trasformazione RFT-to-GSPN

La trasformazione che da Repairable Fault Tree genera Generalized Stochastic Petri Nets, è soggetta alla modularizzazione descritta precedentemente come la Figura 26 illustra. Nella fattispecie sfruttando la relazione di ereditarietà tra i linguaggi FT ed RFT possiamo pensare di progettare la nostra trasformazione attraverso la costruzione di due trasformazioni più semplici: *ft2gspn* e *rft2gspn*.

Figura 26: RFT-to-GSPN: schema di riferimento

Analizziamo le due trasformazioni separatamente.

La trasformazione **ft2gspn** si fonda su una formalizzazione ben nota in letteratura [Bobbio2]; il contributo originale della tesi in tale senso è costituito dalla definizione delle regole dichiarative implementati la trasformazione stessa. Di seguito sono rappresentate le regole di traduzione implementate in ATL di tale trasformazione (si noti che la semplicità della trasformazione è stata tale da non richiedere la scrittura di regole imperative). Nella fattispecie sono rappresentate le traduzioni dei seguenti costrutti FT: BasicEvent (Figura 27), Event connesso a gate di tipo KooN (Figura 28), Event connesso a gate di tipo OR (Figura 29), Event connesso a gate di tipo AND (Figura 30), arco entrante in un gate di tipo KooN (Figura 31), arco entrante in un gate di tipo OR (Figura 32) e arco entrante in un gate di tipo AND (Figura 33).

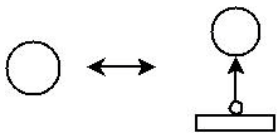

Figura 27: BasicEvent - traduzione in GSPN

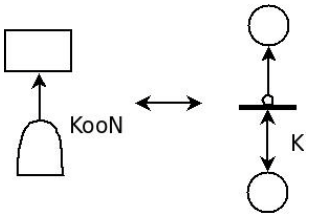

Figura 28: Event con gate KooN - traduzione in GSPN

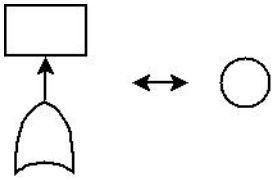

Figura 29: Event con gate OR - traduzione in GSPN

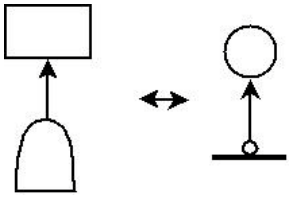

Figura 30: Event con gate AND - traduzione in GSPN

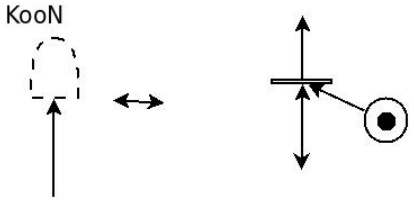

Figura 31: Arco entrante in KooN - traduzione in GSPN

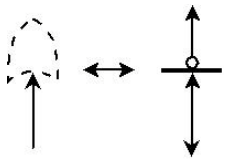

Figura 32: Arco entrante in OR - traduzione in GSPN

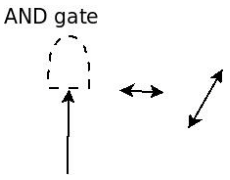

Figura 33: Arco entrante in AND - traduzione in GSPN

Si riportano a titolo di esempio alcune di queste regole in ATL. Innanzitutto la regola relativa al BasicEvent (Figura 27).

```
rule basic {
      from
            e: RFT!BasicEvent
      to
            t: GSPN!Stochastic (name<-e.name->concat('_t'), rate<-e.rate),
            p: GSPN!Place (name<-e.name->concat('_p'), marking<-0),
            a: GSPN!Post (weight<-1, src<-t.name, dest<-p.name),
            i: GSPN!Inhibitor (weight<-1, src<-p.name, dest<-t.name)
}
```

e ad esempio quella relativa ad un middle event con in ingresso un gate di tipo AND (Figura 30):

```
rule andGate {
      from
            e: RFT!MiddleEvent (thisModule.isAND(thisModule.getInGate(e)))
      to
            t: GSPN!Immediate (name<-e.name->concat('_t'), priority<-1),
            p: GSPN!Place (name<-(e.name->concat('_p')), marking<-0),
            a: GSPN!Post (weight<-1, src<-t.name, dest<-p.name),
            i: GSPN!Inhibitor (weight<-1, src<-p.name, dest<-t.name)
}
```

La trasformazione *rft2gspn* è più complessa della precedente. Lo schema di principio è rappresentato in Figura 34.

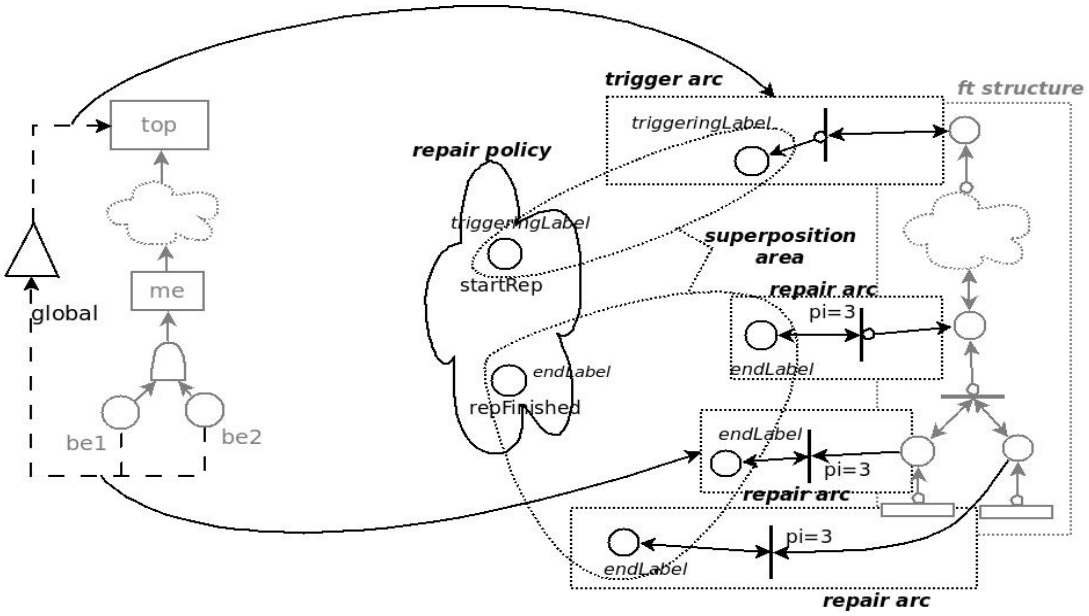

Figura 34: rft2gspn: schema di funzionamento

La trasformazione consta essenzialmente di due matched rules: una sugli archi che vanno dalla Repair Box ai top event (i *trigger arc*) e l'altra dagli archi che vanno dai Basic Event alla Repair Box (i *repair arc*). Si ricorda che l'obiettivo di questa trasformazione, così come descritta all'interno del processo trasformazionale per la riparabilità, è quello di generare il modello GSPN del RFT ad esclusione della politica di riparazione, ossia della sottorete GSPN interna alla RepairBox.

Al fine di poter, però, definire una struttura GSPN che si possa connettere alla rete implementante la suddetta politica, è necessario supporre una sorta di "interfaccia" della politica di riparazione. Nello schema in Figura 34, tale interfaccia è rappresentata dalla nuvola al centro ed è costituita da due posti: uno *startRep* (labellato con etichetta *triggeringLabel*) ed un *repFinished* (labellato con *endLabel*). Il primo posto attiva il meccanismo di riparazione (contenuto nella politica) mentre il secondo ne rappresenta la fine e quindi il momento in cui l'intero Fault Tree (e conseguentemente la rete GSPN) viene "liberato" dai guasti.

Passiamo adesso a descrivere tecnicamente come sono strutturate le due regole.

La prima regola viene innescata da un *trigger arc* ed ha come effetto la semplice creazione della sottorete evidenziata in Figura 34. La regola ATL che implementa questo passaggio é molto semplice e realizzabile attraverso codice dichiarativo.

```
rule trigger {
  from
    a: RFT!TriggerArc
  using {
    rb: RFT!RepairBox = thisModule.getRB(a.src);
  }
  to
    p: GSPN!Place (
      name<-rb.name->concat('_handle'),
      marking<-0,
      label<-'triggeringLabel'
    ),
    t: GSPN!Immediate (name<-rb.name->concat('_trig'), priority<-1),
    pt: GSPN!Pre (weight<-1, src<-a.dest->concat('_p'), dest<-t.name),
    tp: GSPN!Post (weight<-1, src<-t.name, dest<-a.dest->concat('_p')),
    aa: GSPN!Post (weight<-1, src<-t.name, dest<-p.name),
    ii: GSPN!Inhibitor (weight<-1, src<-p.name, dest<-t.name)
}
```

La seconda regola viene innescata da ogni *repair arc* ed è molto più complessa; il suo obiettivo non è solo creare la struttura che "ripari" l'omologo GSPN del BasicEvent collegato al Repair Box dal repair arc, ma anche quello di creare tale struttura di riparazione per ogni evento compreso nel sottoalbero che parte dall'evento scatenante la riparazione e che arriva ai basic events stessi. Questa operazione presenta due problemi tecnici:

esplorazione del sottoalbero: poichè il sottoalbero viene esplorato a partire dalle foglie e poichè si tratta di una matched rule che viene invocata sempre per ogni elemento del modello di partenza che soddisfa la regola, occorre fare attenzione a non visitare più volte lo stesso nodo;

creazione della rete di riparazione ad ogni evento intermedio: la sottorete creata per "riparare" i diversi tipi di eventi (basic, middle con gate in ingresso OR, AND o KooN), sono tra loro diverse e pertanto occorre anche sapere differenziare la traduzione in funzione di tali variazioni.

Il codice ATL che traduce questa matched rule è la seguente:

```
rule repair {
      from
          a: RFT!RepairArc
      do {
            thisModule.rbName<-a.dest;
            thisModule.solve(a.src);
      }
}
```

dove *solve(..)* è una called rule ricorsiva che si occupa di risolvere il primo punto. In particolare per potere esplorare l'intero albero senza ripassare due volte nello stesso punto occorre usare una lista di nodi *visited* implementata in ATL attraverso un attribute helper:

```
helper def : list : Sequence(String) = Sequence{ };
```

Il codice della solve(..) è:

```
rule solve(eventName: String) {
  using {
    e: RFT!Event = thisModule.getEvent(eventName);
    child: RFT!Event = thisModule.getFanOut(e);
    gate: RFT!Gate = thisModule.getInGate(e);
  }
  do {
    -- se l'elemento è già presente nella lista non faccio nulla
    if (not thisModule.list->includes(eventName)) {
      thisModule.list<-thisModule.list->append(eventName);
      -- inizio della valutazione del tipo
      if (e.oclIsKindOf(RFT!BasicEvent)) {
        thisModule.simple(e);
      } else {
        -- è un Middle Event: a che tipo di Gate è collegato?
        if (thisModule.isKooN(gate)) {
          -- per il KooN si avvia la procedura complessa
          thisModule.complex(e);
        } else {
          -- evento collegato ad una porta di tipo OR oppure AND
          thisModule.simple(e);
```

```
      }
    }
    -- Ricorsione
    if ((not thisModule.isRBConnected(e))and(not child.oclIsUndefined()))
      thisModule.solve(child.name);
  }
 }
}
```

Questa called rule è anche il punto di risoluzione del secondo problema, infatti ogni invocazione della solve si interroga sul tipo (*oclIsKindOf*) del evento che sta "visitando" ed invoca a seconda dei casi due tipi di regole (called rule): *simple* se l'evento è di tipo basic, middle con gate OR o middle con gate AND, oppure *complex* se si tratta di un middle event collegato ad un gate di tipo KooN.

Viene riportato il codice della called rule *simple*.

```
rule simple (e: RFT!Event) {
    to
        p: GSPN!Place (name<-e.name->concat('_clearp'), marking<-0,
label<-'endH'),
       t: GSPN!Immediate (name<-e.name->concat('_clear'), priority<-3),
        flushArc: GSPN!Pre (weight<-1, src<-e.name->concat('_p'), dest<-
t.name),
        enabling1: GSPN!Post (weight<-1, src<-t.name, dest<-p.name),
        enabling2: GSPN!Pre (weight<-1, src<-p.name, dest<-t.name)
}
```

4.3.3 La Trasformazione GSPN2Great

Questa trasformazione è stata realizzata attraverso una query atl di cui se ne riportano alcune porzioni. Innanzitutto notiamo che in GreatSPN una rete GSPN viene rappresentata attraverso una coppia di file di testo: un *.def* ed un *.net*. Il primo contiene una stringa costante (in realtà non è significativo per le GSPN ed è conservato solo per compatibilità con il caso più complesso delle SWN).

```
query gspn2def =
    let filename: String = 'rbc.def' in
    '|256\n%\n|\n'.writeTo(filename);
```

Questa query non fa altro che scrivere le stringhe:

```
|256
%
|
```

all'interno del file rbc.def. Più complessa è la struttura del .net che contiene la definizione della rete vera e propria. Questa complessità si riflette in una maggiore articolazione della query:

```
query gspn2net =
      let filename: String = rbc.net' in
      let base : Sequence(Integer) =
Sequence{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20} in
      let   places: Sequence(GSPN!Place) = GSPN!Place.allInstances()-
>asSequence() in
      let maxPriority: Integer = GSPN!Immediate.allInstances()->collect(p |
p.priority)->asSequence()->last() in
      let groups: Sequence(Integer) = base->select(i | i <= maxPriority) in
      let transitions: Sequence(GSPN!Transition) =
GSPN!Transition.allInstances()->asSequence() in
      (
            thisModule.header +
            thisModule.summary(places,transitions,groups) +
            places->collect(x | x.toGreat())->sum() +
            groups->collect(x | x.toGreat())->sum() +
            transitions->collect(x | x.toGreat())->sum()
      ).writeTo(filename);

uses gspn2greatLib;
```

Come si nota dallo statement finale, questa query importa una libreria dove sono contenuti diversi helper richiamati dalla query stessa. Il cuore della query è costituito dalla somma (concatenazione) di strighe secondo il formato di GreatSPN (*thisModule.header* + ...). Sulla base di collezioni di elementi (omogenei) del modello GSPN (come ad esempio places che sono una sequenza dei posti del modello da tradurre). Un esempio di helper è costituito dalla funzione *toGreat* nel contesto di un place. Questo helper non fa altro che tradurre ogni singolo place della rete (ossia ogni elemento appartenente alla sequenza di elementi di places) nel formato desiderato:

```
helper context GSPN!Place def: toGreat() : String =
      self.name + self.getLabel() + ' ' + self.marking.toString() +
'1.000000 1.200000 1.000000 1.200000 0\n';
```

Per ulteriori approfondimenti sul format dei file GreatSPN [Greatspn].

4.4 Il linguaggio ATL

Segue adesso una brevissima panoramica sul linguaggio ATL, necessaria alla comprensione dei dettagli tecnici esposti in questo capitolo [ATL]. Il linguaggio di trasformazione Atlas Trasformation Language (ATL) è sviluppato come una parte della piattaforma ATLAS Model Management Architecture (AMMA). Quest'ultimo costituisce una risposta alla MOF/QVT RFP del OMG per la definizione di uno standard per i linguaggi di trasformazione dei modelli e si focalizza principalmente sulle trasformazioni M2M. ATL, dal canto suo, riesce a supportare anche le trasformazione di modello in testo (M2T). Il motore di trasformazione di ATL transformation fornisce supporto sia per il Meta Object

Facilities (MOF 1.4) definito da OMG che per il meta-meta-modelo Ecore definito dal Eclipse Modelling Framework (EMF).

ATL è un linguaggio di modellazione ibrido in quanto permette la specifica di trasformazioni complesse (attraverso costrutti sia dichiarativi che imperativi).

Una trasformazione di modelli espressa in ATL può essere costituita da uno o più unità:

- nel caso di trasformazioni M2M si ricorre a moduli che contengono le regole di trasformazione vera e propria;
- altri tipi di unità sono le librerie che fungono da collettori di funzioni di supporto alla definizione delle regole dette helper;
- nel caso di trasformazioni M2T si ricorre ad unità che prendono il nome di query. Lo scopo di una query è generare un valore semplice quale una stringa o un intero a partire da modelli sorgente. Questo artefatto potrebbe essere un programma in un linguaggio come Java o C++, una documentazione in formato PDF o in formato HTML, oppure una specifica per un altro tool di linguaggi.

Un modulo ATL si compone dei seguenti elementi:

- una sezione di intestazione (header section) che definisce alcuni attributi relativi al modulo di trasformazione;
- una sezione opzionale di importazione (optional import section) che consente di importare alcune librerie ATL esistenti;
- una serie di helpers;
- una serie di regole (rules) che definiscono il modo in cui vengono generati i modelli target da quelli sorgenti.

Header section

Definisce il nome del modulo di trasformazione e il nome delle variabili corrispondenti ai modelli sorgente e di destinazione. La sua sintassi é la seguente: *module module_name*. Affinché la trasformazione possa funzionare è necessario che il nome del file ATL contenente il codice corrisponda al nome di questo modulo.

La dichiarazione del modello destinazione è introdotta dalla parola chiave create, mentre il modello sorgente è introdotto sia dalla parola chiave *from* (nella modalità in cui la trasformazione genera una copia distinta del modello di uscita) o *refines* (quando la trasformazione sta operando direttamente sul modello in ingresso e lo modifica).

create output_models [from|refines] input_models

La dichiarazione di un modello, sorgente o destinazione, deve essere conforme al regime nome_modello: metamodel_nome. Si noti che il nome dei modelli dichiarati sarà utilizzato per l'identità di ciascuno. Di conseguenza, ogni nome di modello dichiarato deve essere unico all'interno del set di modelli (sia in input che in output). Nel caso della trasformazione RFT2GSPN si avrà:

module rft2gspn

create OUT: GSPN from IN: RFT

Import section

La sezione opzionale di importazione permette di dichiarare quali librerie ATL devono essere importate. La dichiarazione di una libreria ATL si ottiene come segue:

uses extensionless_library_file_name

Helpers

Gli helpers ATL definiscono operazioni e attributi. Possono essere visti come l'equivalente ATL dei metodi Java: essi infatti consentono una fattorizzazione del codice ATL e possono essere chiamati da diversi punti di una trasformazione ATL. A titolo di esempio, è possibile prendere in considerazione un helper che restituisce il massimo di due valori interi:

helper context Integer def : max(x : Integer) : Integer = ...;

Quando l'helper è considerato in un contesto di default (che corrisponde al modulo ATL che lo include) si omette il termine *context* della definizione. Il linguaggio ATL rende anche possibile la definizione di attributi detti *attribute helper* (una sorta di variabili statiche di modulo).

Rules

Le regole di trasformazione costituiscono il costrutto base in ATL per esprimere le trasformazioni logiche. Ci sono fondamentalmente tre tipi di regole: le matched rules, le called rules e le lazy rules. Le matched rules sono utili per specificare quali elementi del modello sorgente devono essere trasformati, il numero e il tipo di elementi del modello destinazione da generare e il modo in cui i detti elementi devono essere inizializzati. Queste regole sono usate con un approccio di tipo dichiarativo. Le called rules rappresentano la parte imperativa del linguaggio ATL, esse vengono invocate da una mathed rule o da un'altra called. Esse, al contrario degli helpers, possono generare gli elementi del modello di destinazione come le matched rules. Per quanto riguarda le lazy rules, esse sono regole

dichiarative ma che devono venire esplicitamente invocate (chiamate) all'interno di matched rules.

Capitolo V
Un caso di studio reale: il Radio Block Centre

5.1 Descrizione del sistema

In questo capitolo si applicherà la metodologia di modellazione ed analisi, da noi proposta nei capitoli precedenti al caso di studio del sistema di segnalamento ferroviario ERTMS/ETCS che costituisce un caso esemplare di sistema complesso e critico [UIC]. Dopo una descrizione dell'architettura di riferimento del sistema e l'indicazione dei requisiti RAMS cui il sistema è soggetto, si descriverà la metodologia di modellazione.

Il sistema di segnalamento ferroviario ERTMS/ETCS è costituito dalle componenti di seguito descritte:

- Radio Block Centre (RBC): è il sottosistema di terra che controlla il movimento di un insieme di treni che attraversano l'area sotto la sua supervisione. Esso comunica con il treno ed ha il compito di gestire correttamente il distanziamento tra i treni fornendo al treno stesso un'autorizzazione al movimento (Movement Authority - MA);
- SottoSistema di Bordo (SSB): esso si trova a bordo del treno ed è deputato a gestire, sulla base delle informazioni ricevute dal RBC, la velocità e lo spazio massimo percorribile e di attuare, qualora tali requisiti di movimento non venissero rispettati, la frenatura del mezzo;
- l'Interlocking (IXL): è il responsabile dell'integrità della via e provvede a creare le condizioni di marcia (itinerari ed attuazione degli enti fisici della via). Al fine di conservare la compatibilità con i sistemi e le regole di segnalamento ferroviario nazionali, lo standard non copre tali sistemi ma unicamente il protocollo di comunicazione tra il RBC e gli IXL;
- la rete di comunicazione (GSM-R): è deputata a provvedere alla gestione della comunicazione tra il sottosistema di bordo e l'RBC attraverso l'adattamento di una normale rete GSM.

Una visione di insieme di tale sistema è raffigurata in Figura 35.

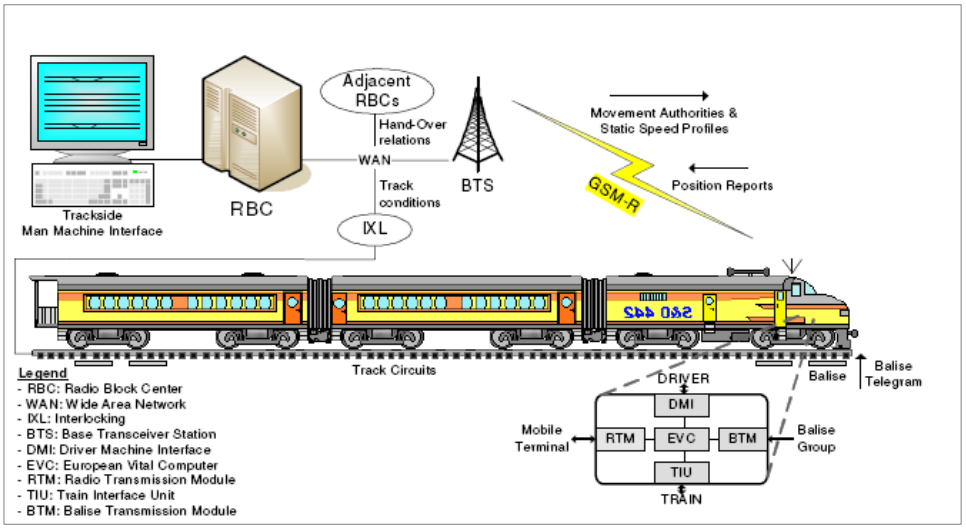

Figura 35: ERTMS/ETCS - visione d'insieme

Oltre ad una descrizione statica provvederemo anche a illustrare i meccanismi di consegna della MA da parte del RBC verso il SSB. Lo standard ERTMS/ETCS non descrive in maniera vincolante la dinamica dell'interazione tra il RBC e il SSB ma mette a disposizione dei fornitori di tecnologie, alcuni strumenti utili nella determinazione di tale interazione. Il RBC invia periodicamente una MA al SSB attraverso la rete GSM-R. Qualora il SSB non riceva tale messaggio per più di un certo periodo, ritiene di aver perso il collegamento con il sistema di terra e provvede ad una frenatura di servizio. Altra causa di tale frenatura è la perdita di connessione radio, l'instaurazione di una sessione di comunicazione GSM-R, che avviene per motivi legati all' indisponibilità delle apparecchiature non vitali della rete in esame (BTS, MSC, Router, …). Il SSB d'altra parte, prima di dichiarare la perdita di collegamento con il sottosistema di terra provvede a ricontattare il RBC per un numero di volte fissato. Qualora tutti i tentativi non vadano a buon fine, esso è costretto a provvedere alla marcia degradata evidenziando così il fallimento del sistema di segnalamento ferroviario ERTMS/ETCS. Il sistema appena descritto rientra nella categoria di sistemi critici perché una indisponibilità delle sue componenti fondamentali potrebbe essere causa di perdita di vite umane e di danni a diversi livelli. Il caso di studio è in particolare costituito da uno dei sottosistemi del sistema ERTMS/ETCS: il Radio Block Centre. Questo sistema costituisce a tutti gli effetti il collo di bottiglia dell'affidabilità e della sicurezza del sistema ERTMS/ETCS intero.

5.1.1 RBC

Il Radio Block Centre è fondamentalmente un sistema a calcolatore costituito da due categorie di componenti. Nella prima rientrano i componenti commerciali tipici di un sistema di elaborazione embedded:

- *TMR*: il centro delle elaborazioni del RBC è costituito da un sistema TMR (Triple Redundunant Module) composto da tre schede di calcolo con votazione a maggioranza 2oo3;
- *Voter*: componente che confronta l'output di diverse CPU e decide, in base alla concordanza tra tali output, qual è l'output del sistema TMR e se ci sono schede di elaborazione in errore che devono essere escluse dall'elaborazione a seguito di errori;
- *BUS*: bus di backplane che collega i diversi sottosistemi e permette lo scambio di dati tra gli stessi;
- *PS* (Power Supply): alimentazione dell'intero sistema di elaborazione;
- *GSM-R*: scheda di interfaccia verso la rete GSM-R per la comunicazione con il sottosistema di bordo;
- *WAN*: scheda di interfaccia con rete di comunicazione fissa al fine di permettere ad RBC lo scambio di messaggi con gli IXL e con gli RBC adiacenti.

Nella seconda categoria ci sono i componenti dedicati ERTMS/ETCS alla gestione di specifici compiti:

- *RTM* (Radio Terminal Module): comunicazione GSM-R con il RBC;
- *DMI* (Driver Machine Interface): interfacciamento input/output verso il macchinista;
- *BTM* (Balise Trasmission Module): ricezione dei telegrammi provenienti dai Punti Informativi;
- *TIU* (Train Interface Unit): interfacciamento verso le apparecchiature fisiche del treno (freni, taglio trazione, etc…).

Una visione di come questi componenti sono organizzati ed interconnessi è data in Figura 36.

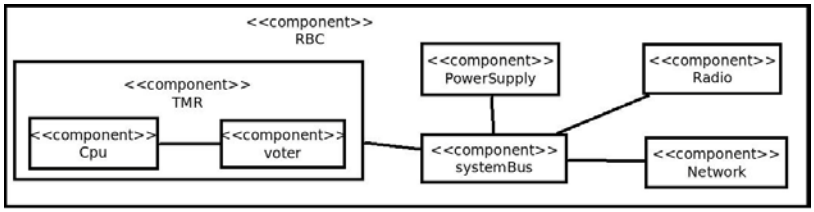

Figura 36: Struttura del RBC

Al fine di rispettare lo standard, RBC deve soddisfare stringenti requisiti RAMS che si pronunciano sulle prestazioni del sistema, soprattutto in termini di affidabilità e di disponibilità [UNISIG]. L'indisponibilità di un RBC è critico poichè non c'è un modo che consenta al sistema di segnalamento di lavorare senza il suo contributo. Nel caso di un malfunzionamento tutti i treni sotto la sua supervisione sono costretti a frenare e a procedere

in modalità controllata dallo staff umano. Ciò porterebbe ai più critici, come già accennato precedentemente, tra i malfunzionamenti ERTMS/ETCS chiamato Immobilising Failure. Lo standard ERTMS/ETCS prevede per il sottosistema RBC un contributo massimo di indisponibilità pari a 10^{-6}.
Questo requisito stringente di disponibilità, unita alla complessità intrinseca del sistema, determina la necessità di studiare non solo il contributo alla disponibilità dell'affidabilità (ossia dalla capacità del sistema di tollerare guasti e dalla qualità dei singoli componenti) ma anche dell'impatto che la manutenzione ha su tale sistema. Questo fa del RBC un valido caso di studio per applicare la metodologia proposta.

5.2 Modello DAM del RBC

Al fine di fornire una descrizione del RBC in termini del profilo MARTE-DAM si è ricorsi all'utilizzo di uno use case diagram, di un component diagram e di una serie di state chart diagram, uno per ogni componente riparabile.

Il diagramma use case è stato utilizzato per descrivere la principale funzionalità del RBC ossia la *Train Outdistancing*. Quest'ultima è stata stereotipata con lo stereotipo *DaService* per indicare, mediante il tagged value *usedResource*, la componente del sistema che fornisce la funzionalità considerata ossia la RBC e mediante il tagged value *ssAvail* il requisito di disponibilità richiesto al servizio. Uno degli attori dello use case (quello coinvolto nel processo di modellazione ed analisi di dependability) è stato stereotipato come *DaAgentGroup* al fine di modellare, mediante il tagged value *skillType* il tipo di competenza richiesta e mediante *agentNumber*, il numero delle risorse richieste per eseguire le attività di riparazione (Figura 37). Si noti che non è stata considerata in tale sede la presenza di manutenzione imperfetta (*correctness* di *DaAgentGroup* pari ad 1.0).

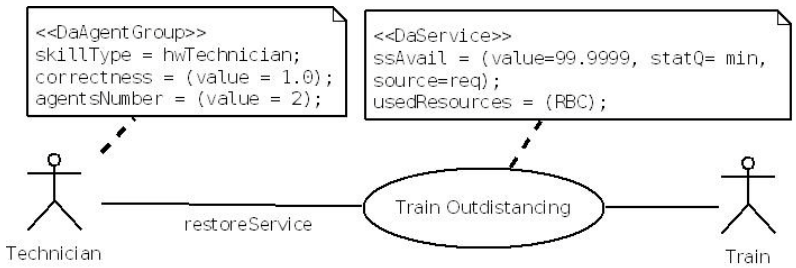

Figura 37: RBC – use case diagram

Il diagramma delle componenti (Figura 38) è stato utilizzato per fornire una descrizione strutturale di alto livello del RBC in cui sono evidenziati gli stereotipi associati sia alle componenti singole (*MainBus*) che a quelle caratterizzate da una struttura interna (*TMR*).

Ogni sottosistema ridondante è stato rappresentato da un package stereotipato come *DaRedundantStructure* (ad es. *SystemBus*) in cui sono state stereotipate come *DaComponent* gli elementi operativi (ad es. *MainBus*) e come *DaSpare* gli elementi di riserva (*SpareBus*).

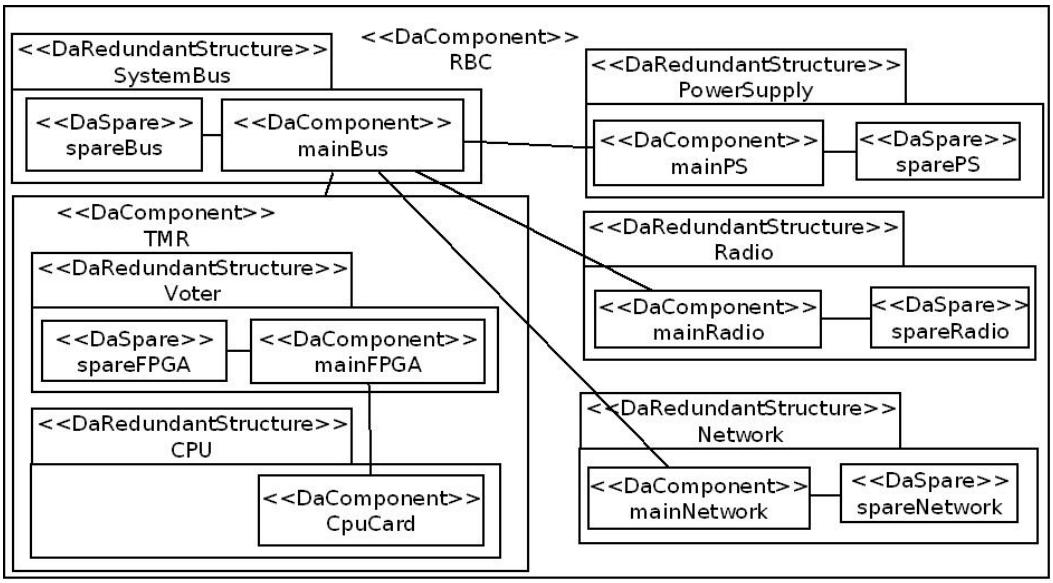

Figura 38: RBC – component diagram

La Figura 39 mostra in dettaglio la porzione del diagramma di Figura 38 relativo al *SystemBus*. Questo livello di dettaglio è necessario per capire il modo in cui i singoli componenti del modello sono stati stereotipati e taggati:

- il tagged value *FTlevel* dello stereotipo *DaRedundantStructure* specifica il numero di risorse minime sia principali che di ricambio per garantire il servizio;
- il tagged value *resMult* degli stereotipi *DaComponent* e *DaSpare* indica rispettivamente il numero delle istanze *mainBus* e di quelle *spareBus*;
- il tagged value *substitutesFor* dello stereotipo *DaSpare* indica il componente che in caso di guasto viene sostituito (nello specifico del sottosistema considerato coincide con il *MainBus*);
- i tagged value *fault.occurenceRate* e *repair.MTTR* sono caratteristiche aggiuntive (rispettivamente i tassi di guasto ed i tempi medi di riparazione) delle componenti stereotipate.

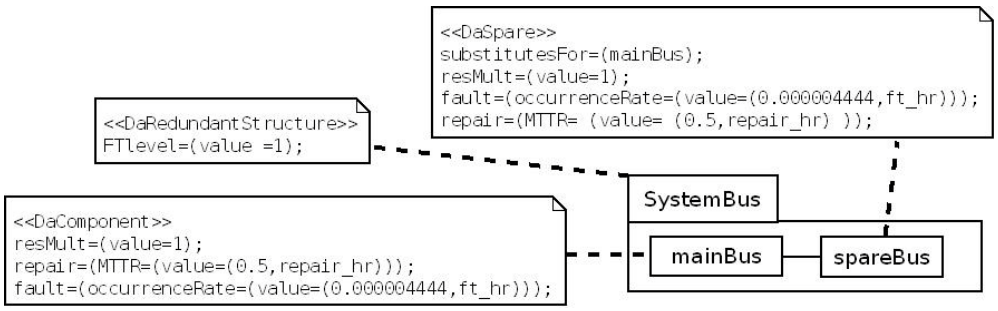

Figura 39: RBC – component diagram (dettaglio)

Gli statechart diagrammi inclusi nella specifica del modello *RBC* sono tre, una per ogni componente riparabile: l'intero *RBC*, il gruppo del *Power Supply* e le *CpuCard*. Data la similarità tra questi diagrammi ne illustreremo solo uno (le *CpuCard* in Figura 40). Tale specifica modella il processo di riparazione della *CpuCard* indicandone i singoli step. In particolare abbiamo considerato un semplice processo di sostituzione che richiede le seguenti transizioni:

- una prima transizione stereotipata come *DaStep* con tagged value di tipo (*kind*) pari a failure ci indica che il componente cui il diagramma è connesso è fallito;
- usa seconda transizione ci indica l'inizio della fase di riparazione. Questa transizione è stereotipata come *DaActivationStep* ed indica tra l'altro il componente che causa l'attivazione (*cause*), il numero di agenti coinvolti (*agentsNumber*) e la specializzazione degli agenti (*agentsSkill*);
- ultima transizione (la *DaReplacementStep*) modella la sostituzione fisica del componente indicato (*replace*).

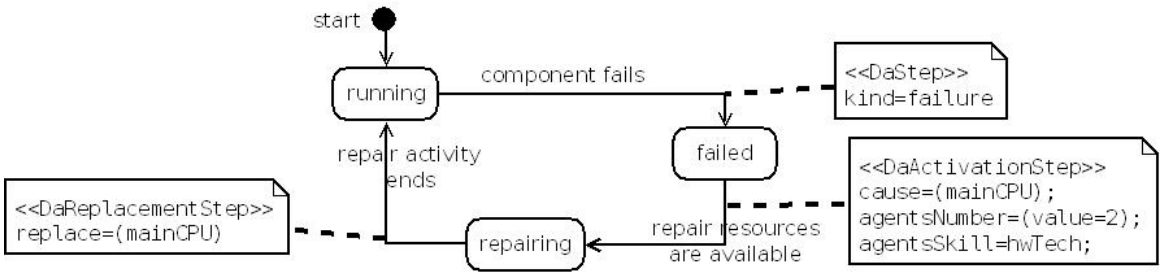

Figura 40: RBC – statechart diagram

5.3 Modello multiformale

Oltre alla specifica data nel paragrafo precedente, dobbiamo aggiungere la specifica (fatta direttamente in GSPN per le motivazioni addotte al Paragrafo 6.3) della politica di riparazione. Su questi due modelli, quello MARTE-DAM dell'intero sistema e la GSPN che modella la politica di riparazione (vedi Figura 41), è stata applicata la transformation chain descritta precedentemente.

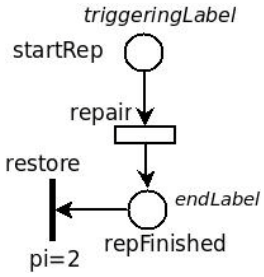

Figura 41: Politica di riparazione

Un primo risultato è costituito dal modello multiformale RFT del RBC ottenuto attraverso l'applicazione della trasformazione DAM-to-RFT. Questo modello è rappresentato in Figura 42.

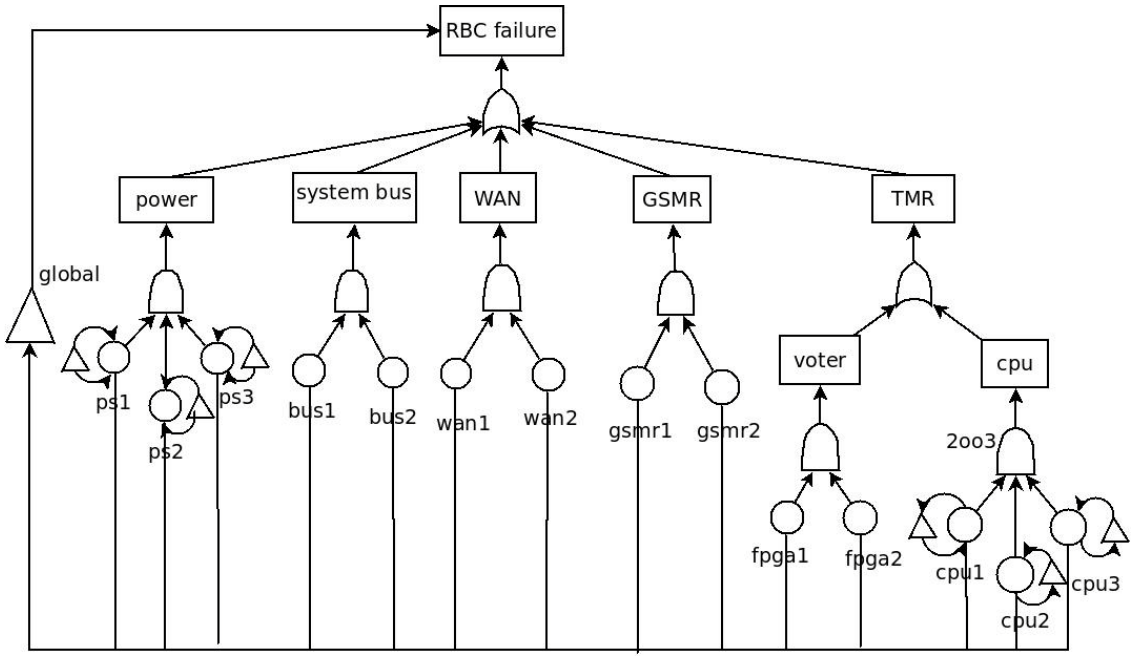

Figura 42: RBC – modello RFT generato

Successivamente questo modello viene tradotto in GSPN e fusa con il modello GSPN in Figura 41. Data la complessità del modello non è possibile raffigurarlo graficamente. Il risultato di questo modello è poi tradotto in formato proprietario GreatSPN ed è stato poi analizzato con il tool GreatSPN.

Ulteriori analisi e studi di sensitività su tale sistema sono state pubblicate in [Flammini].

Capitolo VI
Conclusioni e sviluppi futuri

Questa Tesi ha definito un nuovo approccio alla modellazione ed analisi di sistemi complessi basata sul tecniche e strumenti di Model Driven Engineering e che avessero come obiettivo la generazione di modelli multiformali di sistemi critici e complessi.

I contributi originali portati da questa tesi sono: (1) definizione di una **nuova metodologia** in grado di coniugare i vantaggi dei principali approcci esistenti in letteratura sull'uso dei metodi formali nello studio di sistemi critici (generazione automatica di modelli e multiformalismo); (2) focus della tesi sul tema della **riparabilità** (molto sentito in ambito industriale ma forse non sufficientemente studiato in ambito accademico) (3) **estensione di MARTE-DAM** nell'ottica di un perfezionamento di tale profilo alla modellazione di strutture ridondate fault tolerant (KooN, AND, OR, etc...) e di processi complessi di riparazione (4) definizione di tecniche per la **progettazione incrementale di trasformazioni** tra modelli che tengano conto delle relazioni di ereditarietà tra linguaggi (5) applicazione delle tecniche e delle metodologie definite al caso di studio reale del **Radio Block Centre** del sistema ERTMS/ETCS, sistema di interesse industriale nell'ambito del dominio del segnalamento ferroviario.

Questa Tesi è comunque un punto di partenza per ulteriori ricerche volte, innanzitutto, a completare l'implementazione dell'architettura di riferimento per l'automatizzazione della metodologia ma, specialmente, ad ampliare e perfezionare la metodologia stessa prevalentemente attraverso: nuovi casi di studio anche legati a sistemi software intensive e applicazione della metodologia all'analisi di ulteriori attributi di dependability (safety, performability, etc...).

Bibliografia

[Ajmone] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis: *Modelling with Generalized Stochastic Petri Nets*. J. Wiley and Sons ed., 1995

[ATL] ATL user manual version 7 February 2006 by ATLAS group LINA & INRIA Nantes

[Avizienis] Avizienis, J. C. Laprie, B. Randel, *Fundamental Concepts of Dependability*. Research Report N01145, LAAS-CNRS. April 2001.

[Ascher] H. Ascher and H. Feingold, *Repairable Systems Reliability*, Marcel Dekker, Inc., New York, 1984.

[Barlow] R.E. Barlow and F. Proschan, *Mathematical Theory of Reliability*, John Wiley & Sons, Inc., New York, 1965.

[Beizer] Beizer, Boris, *Software Testing Techniques*, 2nd edition, New York: Van Nostrand Reinhold, 1990.

[Bernardi] Simona Bernardi , José Merseguer , Dorina C. Petriu, *Adding Dependability Analysis Capabilities to the MARTE Profile*, Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, September 28-October 03, 2008, Toulouse, France

[Bernardi2] Bernardi S., Flammini F., Marrone S., Merseguer J., Papa C., Vittorini V., Model-driven availability evaluation of railway control systems In Computer Safety, Reliability, and Security 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-21, 2011, Lecture Notes in Computer Science Vol. 6894, Pag. 15-28, ISBN: 978-3-642-24269-4, Springer-Verlag, 2011

[Bobbio] A. Bobbio, L. Portinale, M. Minichino, E. Ciancamerla, *Improving the Analysis of Dependable Systems by Mapping Fault Trees into Bayesian Networks*, Reliability Engineering and System Safety Journal – 71/3 – pp 249-260, 2001.

[Bobbio2] Bobbio, A., Franceschinis, G., Gaeta, R., Portinale, L.: Parametric Fault Tree for the Dependability Analysis of Redundant Systems and Its High-Level Petri NetSemantics. IEEE Transaction on Software Engineering. vol. 29(3), p. 270-287 (2009)

[Bondavalli1] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza and G. Savoia, “Dependability Analysis in the Early Phases of UML Based System Design”, International Journal of Computer Systems –Science & Engineering, Vol. 16 (5), Sep 2001, pp. 265-275.

[Bondavalli2] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. “Dependability analysis in the early phases of UML based system design”. Journal of Computer Systems Science and Engineering, 16(5):265-275, 2001.

[Cassady] C.R. Cassady, E.A. Pohl, W.P. Murdock. *Selective Maintenance Modeling for Industrial Systems*. In Journal of Quality in Maintenance Engineering, Vol. 7, No. 2, 2001, pp. 104-117.

[CENELEC1] CENELEC EN 50129: *Railway applications - Communication, signalling and processing systems* - Safety related electronic systems for signalling, 2003.

[CENELEC2] CENELEC EN 50126: *Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*, 2001.

[Charniak] Eugene Charniak, *Bayesian Networks without Tears*, AI Magazine, 1991

[Chess] ARTEMIS-JU-100022 CHESS - Composition with guarantees for High-integrity Embedded Software components aSsembly. http://www.chess-project.org.

[Ciardo] G. Ciardo, R.L. Jones, A.S. Miner, R. Siminiceanu. *SMART: Stochastic Model Analyser for Reliability and Timing*. Tools of International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems. Aachen – Germany. Sept. 2001

[Codetta] Codetta Raiteri, D., Franceschinis, G., Iacono, M., Vittorini, V., *Repairable Fault Tree for the Automatic Evaluation of Repair Policies*, In IEEE Proceedings of the International Conference on Dependable Systems and Networks (DSN'04), Florence, Italy, June 28-July 1, 2004: p. 659.

[DalCin] M. Dal Cin, G. Huszerl, K. Kosmidis, "Evaluation of Safety-Critical Systems Based on Guarded Statecharts", In A. Williams, editor, Proc. Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE'99), IEEE Computer Society Press, 1999.

[D'Ambrogio] A. D'Ambrogio, G. Iazeolla, R. Mirandola. "A method for the prediction of software reliability", In Proc. of the 6th IASTED Software Engineering and Applications Conference (SEA'02), 2002.

[deLara] J. de Lara, H. Vaunghelewe. *AToM^3: A Tool for Multi-formalism Modelling and Meta-modelling*. In Proc. Of the European Joint Conference on Theory and Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE). Sprinter-Verlag. April 2002. Granoble. France

[Delic] K. A. Delic, F. Mazzanti, L. Strigini, *Formalizing Engineering Judgement on Software Dependability via Belief Networks*, 6th IFIP Working Conference on Dependable Computing for Critical Applications, 1997

[Duffy] David A Duffy, *Principles of Automated Theorem Proving*. John Wiley & Sons, (1991).

[Clarke] E. M. Clarke, Jeannette M. Wing: *Formal Methods: State of the Art and Future Directions*. ACM Comput. Surv. 28(4): 626-643 (1996)

[Deavours] D. Deavours, G. Clark, T. Courteney, D. Daly, S. Derisavi, J.M. Doyle, W.H. Sanders, P.G. Webster. *The Möbius Framework and its Implementation*. IEEE Transactions on Software Engineering, 28(10). 2002.

[DiLorenzo] G. Di Lorenzo, F. Flammini, M. Iacono, S. Marrone, F. Moscato, V. Vittorini, The software architecture of the OsMoSys Multisolution Framework, In Proc.of 2nd

International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS), Nantes (France) - October 2007 – ISBN: 978-963-9799-00-4

[Espinoza] Espinoza, H., Dubois, H., Gerard, S., Medina, J., Petriu, D.C. and Woodside M., *Annotating UML Models with Non-Functional Properties for Quantitative Analysis* in MoDELS 2005 Workshops (Jean-Michel Bruel, Ed.), LNCS 3844, pp. 79--90, Springer-Verlag, 2006.

[Flammini] F. Flammini, M. Iacono, S. Marrone, N. Mazzocca. Using Repairable Fault Trees for the evaluation of design choices for critical repairable systems. In Proc. 9th High Assurance System Engineering (HASE),Heidelberg (Germany). IEEE - ISBN: 0-7695-377-3. October 2005

[Greatspn] GreatSPN 2.0.2 User's Manual. Performance Evaluation Group. University of Turin.

[Heath] W. S. Heath: *Real-Time Software Techniques*. Van Nostrand Reinhold, New York (1991).

[Hsueh] M. Hsueh, T. Tsai, R. Iyer: *Fault injection techniques and tools*. In IEEE Computer, vol. 30, no. 4, April 1997, pp. 75-82.

[Huszerl] G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis, M. Dal Cin, "Quantitative Analysis of UML Statechart Models of Dependable Systems", The Computer Journal, Vol 45(3), May 2002, pp. 260-277.

[Jouault] F. Jouault, I. Kurtev. On the Architectural Alignment of ATL and QVT. In Proceedings of ACM Symposium on Applied Computing (SAC 06), Model Transformation Track. Dijon (Bourgogne, FRA), April 2006 [Kleppe] Kleppe, Warmer, J., Bast., W.: *MDA Explained, The Model-Driven Architecture*.

[Kurtev] I. Kurtev, Klaas van den Berg, Frédéric Jouault: Rule-based modularization in model transformation languages illustrated with ATL. Sci. Comput. Program. 68(3): 138-154 (2007)

[Laprie] J.C. Laprie. *Dependability: Basic Concepts and Terminology, Dependable Computing and Fault-Tolerance.* Springer-Verlag, Vienna, Austria. 1992.

[Marrone] S. Marrone, C. Papa, and V. Vittorini. Multiformalism and transformation inheritance for dependability analysis of critical systems. In Proceedings of 8th Integrated formal methods, IFM'10, pages 215–228, Berlin, Heidelberg, 2010. Springer-Verlag

[Mellor] S.J. Mellor, K. Scott, A. Uhl, D. Weise. *MDA Distilled. Principles of Model Driven Architecture.* Addison-Wesley, 2004.

[Merz] S. Merz, *Model Checking: a tutorial overview.* In: modeling and Verification of Parallel Processes. LNCS, Vol. 2067, pp. 3-38, 2001.

[Montecchi] L. Montecchi, P. Lollini, A. Bondavalli; Towards a MDE Transformation Workflow for Dependability Analysis. ICECCS 2011, pp. 157-166, 2011

[OMG_MARTE] Object Management Group, “A UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems”, version 1.0, November 2009. http://www.omg.org/spec/MARTE/1.0/.

[OMG_QFTP] Object Management Group, “UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms”, version 1.0, February 2008. http://www.omg.org/spec/QFTP/1.0/.

[Pride] PRIDE – Ambiente di PRogettazione Integrato per sistemi DEpendable, Transformations for Dependability Analysis, Deliverable 2.1, February 2003.

[Ross] S.M. Ross, *Introduction to Probability Models*, Seventh Edition, Harcourt Academic Press, San Diego, 1989.

[RTCA] RTCA SC-167, EUROCAE WG-12: *DO-178B / ED-12B - Software Considerations in Airborne Systems and Equipment Certification* (1992).

[Saturn] http://www.saturn-fp7.eu/

[Schmidt]. D. C. Schmidt, “Guest Editor’s Introduction: Model-Driven Engineering”, IEEE Computer, vol. 39, no. 2, pp. 25-31, Feb. 2006.

[Sebastien] Sébastien Demathieu, Thales Research & Technology, ‘‘MARTE Tutorial: An OMG UML profile to develop Real-Time and Embedded systems”, http://www.omgmarte.org/References.htm, (Sep 2007), Paris.

[Trivedi] K.S. Trivedi. *SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator*. In Proceeding of Dependable Systems and Networks. 2002. ISBN 0-7695-1597-5.

[UIC] UIC, *ERTMS/ETCS class1 System Requirements Specification*, Ref. SUBSET-026, issue 2.2.2, 2002.

[UNISIG] UNISIG, *ERTMS/ETCS RAMS Requirements Specification*, Ref. 96s1266.

[FThandbook] U. S. Nuclear Regulatory Commission, *Fault Tree Handbook*, NUREG-0492, 1981

[Vittorini] V. Vittorini, M. Iacono, N. Mazzocca, G. Franceschinis. *The OsMoSys approach to multiformalism modeling of systems*. In Journal of Software and Systems Modeling. Volume 3(1): 68-81. March 2004.

[Walter] M. Walter, C. Trinitis, W. Karl, “OpenSESAME: An Intuitive Dependability Modeling Environment Supporting Inter-Component Dependencies”, In Proc. of the 2001 Pacific Rim Int. Symposium on Dependable Computing, pp 76-84, IEEE Computer Society, 2001.

[Wing] J. M. Wing, *A Specifier’s Introduction to Formal Methods*. In IEEE Computer, Vol. 23, No. 9, September 1990, pp. 8-24.