SARDEGNA RICERCHE

Sintesi e Analisi Automatica di CPS

UNIONE EUROPEA
Fondo europeo di sviluppo regionale
REPUBBLICA ITALIANA
REGIONE AUTÒNOMA DE SARDIGNA
REGIONE AUTONOMA DELLA SARDEGNA
POR FESR
SARDEGNA 2014-2020

SARDEGNA RICERCHE

Indice

UNIONE EUROPEA
Fondo europeo di sviluppo regionale
REPUBBLICA ITALIANA
REGIONE AUTÒNOMA DE SARDIGNA
REGIONE AUTONOMA DELLA SARDEGNA
POR FESR SARDEGNA 2014-2020

SARDEGNA RICERCHE

Sommario

La società dipende sempre più da sistemi informatici e software dedicati per l'assistenza in quasi ogni aspetto della vita quotidiana. Spesso non ci rendiamo nemmeno conto che sono coinvolti computer e software. Diverse funzioni di controllo nelle auto moderne sono basate su soluzioni software integrate, ad es. freni, airbag, cruise control e sistemi di iniezione del carburante. Cellulari, sistemi di comunicazione, dispositivi medici, sistemi audio e video e dispositivi elettronici di consumo in generale contengono grandi quantità di software. Anche i sistemi di trasporto, di produzione e di controllo stanno applicando sempre più soluzioni software integrate per ottenere flessibilità e un buon rapporto tra costo ed efficienza. Un esempio comune è la crescente complessità dei sistemi, una tendenza che è accelerata dall'adattamento di soluzioni di rete cablate e wireless: in una moderna automobile ad esempio le funzioni di controllo sono distribuite su più unità di elaborazione che comunicano attraverso reti e bus dedicati. Tuttavia, le soluzioni basate su computer e software sono diventate onnipresenti e si trovano in diversi sistemi critici per la sicurezza. Quindi la sfida principale per il campo dell'informatica è quella di fornire formalismi, tecniche e strumenti che consentano la progettazione efficiente di sistemi corretti e ben funzionanti, nonostante la loro complessità. Negli ultimi due decenni circa un approccio molto attraente verso la correttezza dei sistemi di controllo basati su computer sono quelli della verifica del modello (Model Checking). Il model checking è una tecnica di verifica formale che consente di verificare le proprietà comportamentali desiderate di un determinato sistema sulla base di un modello adeguato del sistema attraverso un'ispezione sistematica di tutti gli stati del modello. L'attrattiva del model checking deriva dal fatto che è completamente automatico e offre controesempi nel caso in cui un modello non riesca a soddisfare una proprietà offrendo informazioni indispensabili per le attività di debug. Inoltre, le prestazioni degli strumenti di verifica dei modelli si sono dimostrate essere mature come testimoniato da un gran numero di applicazioni industriali di successo.

Inoltre vengono proposti i Modelli di Specifica di Proprietà (PSP) per facilitare la formalizzazione dei requisiti e che consentono la verifica automatizzata. In particolare, la consistenza interna delle specifiche scritto con i PSP può essere controllata automaticamente con l'uso, ad esempio, di risolutori di soddisfacibilità per la Logica Temporale Lineare (LTL). Tuttavia, per la maggior parte delle applicazioni pratiche, l'espressività dei PSP è troppo limitata ad abilitare la scrittura di specifiche dei requisiti utili e dimostrare che un insieme di requisiti è incostistente può essere inutile a meno che non venga estratto un insieme minimo di requisiti in conflitto per aiutare i progettisti a correggere una specifica sbagliata. In questo documento, vengono estesi i PSP aggiungendo le asserzioni numeriche atomiche, viene presentata una codifica

da PSP alle formule LTL e un algoritmo che estrae un sottoinsieme inconsistente e irriducibile dell'insieme originale dei requisiti. La nostra estensione ci consente di ragionare sulla consistenza interna dei requisiti funzionali che non sarebbe rilevata dai PSP di base. I risultati sperimentali dimostrano che con questo approccio si può verificare e spiegare (in)consistenze nelle specifiche con quasi duemila requisiti generati usando un modello probabilistico, e che consente un'efficace gestione nei casi di studio reali.

Capitolo 1

Introduzione

Nel contesto dei Sistemi Cyber-Fisici (CPS) critici per la sicurezza e l'incolumità, verificare la ragionevolezza dei requisiti funzionali è un compito indiscutibile, ma impegnativo. I requisiti scritti in linguaggio naturale richiedono tempo e verifiche manuali soggette ad errori, mentre consente la verifica automatica spesso richiede formalizzazioni eccessivamente onerose. Data la crescente pervasività di CPS, i loro rigidi vincoli di "*time-to-market*" e budget di prodotto, sono a disposizione pratiche soluzioni per consentire la verifica automatica dei requisiti. I Modelli di Specifica delle Proprietà (PSP) (vedi Dwyer et al. (1999)) offrono un percorso percorribile verso questo obiettivo. I PSP sono una raccolta di astrazioni di specifiche parametrizzabili, di alto livello e indipendenti dal formalismo, originariamente sviluppate per acquisire soluzioni ricorrenti alle esigenze dell'ingegneria dei requisiti. Ogni modello può essere codificato direttamente in un linguaggio formale delle specifiche, come la *Logica Temporale Lineare* (LTL) (vedi Pnueli and Manna (1992)), *Computational Tree Logic* (CTL) (vedi Clarke et al. (1986)), or *Graphical Interval Logic* (GIL) (vedi Dillon et al. (1994)). A causa delle loro caratteristiche, i PSP possono alleggerire l'onere della formalizzazione dei requisiti, ma consentono il controllo di integrità utilizzando strumenti di ragionamento automatico allo stato dell'arte – vedi, ad esempio, Li et al. (2013a, 2015); Schwendimann (1998); Cimatti et al. (2002); Hustadt and Konev (2003). Il controllo di integrità dei requisiti può consistere di tre parti: controllo di ridondanza (vacuità), controllo di completezza e di consistenza (vedi Barnat et al. (2016)). Informalmente, una specifica è soddisfatta in modo vacuo in un modello se è soddisfatta in qualche modo non interessante. Il controllo della vacuità può anche essere eseguito senza la necessità di un modello e in questo caso à noto come controllo di vacuità *intrinseca* (vedi Rozier and Vardi (2011)). Il controllo della completezza è equivalente a verificare se l'insieme dei requisiti copre tutti i comportamenti ragio-

nevoli di un sistema. La completezza può essere verificata in combinazione con un modello di sistema, ma in Barnat et al. (2016) viene presentata anche una proposta per il controllo della completezza senza modello. Infine, la coerenza dei requisiti consiste nel verificare se un sistema *reale* può essere implementato da un determinato insieme di requisiti. Pertanto, ci sono due tipi di verifica possibile (vedi Rozier and Vardi (2011)): (*i*) *realizzabilità*, cioè testando se esiste un sistema *aperto* che soddisfi tutte le proprietà dell'insieme (vedi Pnueli and Rosner (1989)) e (*ii*) *soddisfacibilità*, cioè testando se esiste un sistema *chiuso* che soddisfa tutte le proprietà dell'insieme. Il controllo della soddisfacibilità garantisce che la descrizione del comportamento di un sistema sia internamente coerente e non sia soggetta a vincoli eccessivi o insufficienti Anche se il test di soddisfacibilità è piè debole del test di realizzabilità, la sua importanza è ampiamente riconosciuta (vedi Rozier and Vardi (2011)). In questo lavoro, limitiamo la nostra attenzione alla verifica dell'integrità come controllo di soddisfacibilità. Parliamo della consistenza (interna) dei requisiti scritti utilizzando PSP tenendo in mente che i PSP possono essere tradotti in formule LTL la cui soddisfacibilità può essere verificata usando metodi e strumenti disponibili in letteratura - si veda, ad esempio, Wolper (1985); Kesten et al. (1993); Schwendimann (1998); Manna and Pnueli (2012) per i metodi basati su tableau e Rozier and Vardi (2007, 2010, 2011); Li et al. (2013b, 2014) per i metodi basati su approcci teorico-automatici. La formulazione originale dei PSP si occupa della struttura temporale rispetto alle variabili booleane, ma per la maggior parte delle applicazioni pratiche tale espressività è troppo limitata. Questo è il caso del controller incorporato per manipolatori robotici che è attualmente in sviluppo nel contesto del progetto UE CERBERO (vedi Masin et al. (2017)). Ad esempio, prendiamo in considerazione la seguente dichiarazione: "*L'angolo del giunto1 non deve mai essere maggiore di 170 gradi*". Questo requisito impone una soglia di sicurezza relativa ad alcune articolazioni del manipolatore (giunto1) rispetto a pose fisicamente realizzabili, tuttavia non può essere espresso come PSP a meno che non aggiungiamo asserzioni numeriche atomiche in qualche sistema di vincoli $\mathcal{D}$. Chiamiamo Constraint PSP, o in breve PSP($\mathcal{D}$), un modello che ha la stessa struttura di un PSP, ma contiene proposizioni atomiche di $\mathcal{D}$. Ad esempio, utilizzando PSP($\mathbb{R}, <, =$) possiamo riscrivere il requisito di cui sopra come un modello di *universalità*: "*Globalmente, è sempre il caso che* $\theta_1 < 170$ *è soddisfatto*", dove θ_1 è il segnale numerico (variabile) associato all'angolo del *giunto1*. In linea di principio, il ragionamento automatico su Constraint PSP può essere eseguito in *Constraint Linear Temporal Logic*, cioè, il linguaggio LTL esteso con asserzioni atomiche da un sistema di vincoli (vedi Demri and D'Souza (2007)): nel nostro esempio, la codifica sarebbe semplicemente $\Box(\theta_1 < 170)$. Sfortunata-

mente, questo approccio non sempre si presta ad una soluzione pratica, perchè il linguaggio Constraint Linear Temporal Logic è in generale indecidibile (vedi Comon and Cortier (2000)). Restrizioni su $\mathcal{D}$ possono ripristinare la decidibilità (vedi Demri and D'Souza (2007)), ma introducono limitazioni nell'espressività dei corrispondenti PSP. In questo documento, proponiamo una soluzione che garantisce che la verifica automatizzata della consistenza è fattibile, ma consente ai PSP di mettere insieme le variabili booleane e i segnali numerici (vincolati). Il nostro approccio ci consente di acquisire molte specifiche di interesse pratico e di scegliere una procedura di verifica dal pool relativamente grande di sistemi automatizzati di ragionamento disponibili per LTL. In particolare, limitiamo la nostra attenzione a un sistema di vincoli della forma ($\mathbb{R}$,<,=) e proposizioni atomiche della forma $x < c$ o $x = c$, dove $x \in \mathbb{R}$ è una variabile e $c \in \mathbb{R}$ è un valore costante. In seguito, denotiamo con $\mathcal{D}_C$ per indicare tale restrizione.

Sapendo che una serie di requisiti scritti con PSP($\mathcal{D}_C$) è (in)consistente è solo il primo passo per scrivere una specifica corretta. In caso di inconsistenza dei requisiti, ottenere un insieme minimo di tali requisiti sarebbe auspicabile per aiutare i progettisti ad evitare i controlli manuali per individuare i problemi in una specifica. Il problema di trovare sottoinsiemi minimi insoddisfacenti, o *spiegazioni dell'inconsistenza*, è stato oggetto di qualche attenzione, ad es. nella soddisfacibilità proposizionale e nella programmazione a vincoli. Gli algoritmi che si trovano in letteratura possono essere specifici del dominio – vedi, ad es. Belov and Marques-Silva (2012); Liffiton and Sakallah (2008) – o indipendente dal dominio – vedi, ad esempio, Junker (2001). Possono essere ulteriormente suddivisi in algoritmi che trovano solo un sottoinsieme incoerente o tutti i sottoinsiemi incoerenti. Per quanto a nostra conoscenza, per il controllo della soddisfacibilità del linguaggio LTL non è stato implementato alcun algoritmo specifico per questo dominio, mentre è stato studiato un algoritmo di uso generale per il calcolo di tutti i nuclei non soddisfacibili. Poiché per motivi pratici nella progettazione dei requisiti è meglio avere una risposta in breve tempo piuttosto che una risposta completa, vi presentiamo un metodo per cercare inconsistenze in modo incrementale, cioè, interrompendo la ricerca una volta che almeno un sottoinsieme di inconsistenza (minimo) è stato trovato. In particolare, dato un insieme di requisiti inconsistenti, ne estraiamo un sottoinsieme minimo (irriducibile) che è ancora inconsistente. L'insieme è garantito essere minimo nel senso che, se rimuoviamo uno degli elementi, l'insieme rimanente diventa consistente.

Nel complesso, il nostro contributo può essere riassunto come segue:

– Estendiamo i PSP di base sul sistema di vincoli $\mathcal{D}_C$

– Forniamo una codifica da qualsiasi PSP($\mathcal{D}_C$) in una formula LTL corrispondente.

– Forniamo uno strumento [1] basato su procedure decisionali e model checker allo stato dell'arte per analizzare automaticamente i requisiti espressi come PSP($\mathcal{D}_C$).

– Proponiamo algoritmi dedicati all'estrazione di sottoinsiemi minimi di requisiti inconsistenti e li implementiamo nello strumento sopra menzionato.

– Implementiamo un generatore di requisiti artificiali espressi come PSP($\mathcal{D}_C$); il generatore prende un insieme di parametri in input ed emette una collezione di PSP secondo un modello di probabilità parametrizzato.

– Usando il nostro generatore, eseguiamo un'esaustiva valutazione sperimentale mirata a comprendere (i) quale strumento di ragionamento automatico è il migliore nel gestire un insieme di requisiti come PSP($\mathcal{D}_C$) e (ii) se il nostro approccio è scalabile.

La spiegazione riguardo la verifica e l'inconsistenza dei requisiti scritti in PSP($\mathcal{D}_C$) viene effettuata utilizzando strumenti e tecniche disponibili in letteratura (vedi Rozier and Vardi (2010, 2011); Li et al. (2013a)). Tramite questi strumenti, dimostriamo la scalabilità del nostro approccio controllando la consistenza fino a 1920 requisiti, con 160 variabili e fino a 8 differenti valori costanti che appaiono in asserzioni atomiche, in meno di 500 secondi di CPU. Un totale di 75 requisiti relativi al controller integrato per il progetto CERBERO vengono verificati in pochi secondi, anche senza ricorrere al miglior strumento tra quelli che consideriamo.

[1] `https://gitlab.sagelab.it/sage/SpecPro`

Capitolo 2

Nozioni di base

In questo capitolo, introduciamo le nozioni di base relative alla verifica dei sistemi e all'analisi dei requisiti. Nello specifico, La Sezione 2.1 riporta una breve introduzione relativa alla verifica dei sistemi. La Sezione 2.2 introduce la verifica del modello (Model Checking). La Sezione 2.3 introduce le caratteristiche del Model Checking. La Sezione 2.4 introduce il linguaggio LTL (Logica Temporale Lineare). La Sezione 2.5 introduce la verifica della soddisfacibilità LTL. La Sezione 2.6 introduce i modelli di specifica della proprietà (PSP).

2.1 Verifica dei sistemi

La nostra fiducia sul funzionamento dei sistemi ICT (Tecnologia della Informazione e della Comunicazione) sta crescendo rapidamente. Questi sistemi stanno diventando sempre più complessi e invadono in maniera massiccia la vita quotidiana attraverso Internet e attraverso tutti i tipi di sistemi integrati come smart card, computer palmari, telefoni cellulari e televisori di fascia alta. Nel 1995 è stato stimato che ci troviamo di fronte a circa 25 dispositivi ICT su base giornaliera. Servizi come le *banche elettroniche* e la teleacquisti sono diventati realtà. Il flusso di cassa giornaliero via Internet è di circa 10^{12} milioni di dollari USA. Circa il 20% dei costi di sviluppo di prodotto dei moderni dispositivi di trasporto come automobili, treni ad alta velocità e aeroplani sono dedicati ai sistemi di elaborazione delle informazioni. I sistemi ICT sono universali e onnipresenti. Controllano il mercato borsistico, costituiscono il cuore dei commutatori telefonici, sono fondamentali per la tecnologia di Internet e sono vitali per diversi tipi di sistemi medicali. La nostra dipendenza dai sistemi integrati rende l'affidabilità di funzionamento una grande importanza sociale.

Oltre ad offrire una buona prestazione in termini di tempi di risposta e capacità di elaborazione, l'assenza di errori fastidiosi è una delle principali indicazioni di qualità.

È tutta questione di soldi. Siamo infastiditi quando il nostro telefono si guasta, o quando il nostro computer reagisce in modo inaspettato o sbagliato rispetto ai comandi emessi. Questi errori software e hardware non minacciano le nostre vite, ma possono avere notevoli conseguenze a livello finanziario per il produttore. I sistemi ICT corretti sono essenziali per la sopravvivenza di un'azienda. Sono noti esempi drammatici. L'errore presente nell'unità di divisione in virgola mobile Pentium II di Intel nei primi anni Novanta causò una perdita di circa 475 milioni di dollari USA per la sostituzione dei processori difettosi e danneggiò gravemente la reputazione di Intel come produttore di chip affidabili. L'errore del software in un sistema di smistamento bagagli posticipò l'apertura dell'aeroporto di Denver per 9 mesi, con una perdita di 1,1 milioni di dollari USA al giorno. Ventiquattro ore di mancato funzionamento del sistema di prenotazione di biglietti online su scala mondiale di una grande compagnia aerea ne causerà il suo fallimento a causa degli ordini mancati.

Figura 2.1: Il lancio di Ariane-5 il 4 giugno 1996; si è schiantato 36 secondi dopo il lancio a causa di una conversione di un valore in virgola mobile a 64 bit in un valore intero a 16 bit.

È tutta questione di sicurezza: gli errori possono essere anche catastrofici. I difetti fatali nel software di controllo del missile Ariane-5 (Figura 2.1), del Mars Pathfinder e degli aerei della famiglia Airbus hanno fatto scalpore sui giornali di tutto il mondo e sono ormai noti. Software simili vengono utilizzati per il controllo di processo di sistemi critici per la sicurezza come impianti chimici, centrali nucleari, sistemi di controllo del traffico e di allerta e barriere anti-tempesta. Chiaramente, gli errori in questi tipi di software possono avere conseguenze disastrose. Ad esempio, un difetto software nella parte di controllo della macchina per radioterapia Therac-25 ha causato la morte di sei pazienti affetti da cancro tra il 1985 e il 1987 in quanto esposti a un sovradosaggio di radiazioni.

La crescente dipendenza delle applicazioni critiche nell'elaborazione delle informazioni ci porta a dichiarare: *L'affidabilità dei sistemi ICT è una questione chiave*

nel processo di progettazione di un sistema.

La portata dei sistemi ICT e la loro complessità aumentano rapidamente. I sistemi ICT non sono più autonomi, ma sono tipicamente incorporati in un contesto più ampio, connettendosi e interagendo con molti altri componenti e sistemi. Diventano quindi molto più vulnerabili agli errori: il numero di difetti cresce in modo esponenziale con il numero di componenti di sistema interagenti. In particolare, fenomeni come la concorrenza e il non determinismo che sono fondamentali per modellare i sistemi interagenti, risultano essere molto difficili da gestire con tecniche standard. La loro crescente complessità, insieme alla pressione per ridurre drasticamente i tempi di sviluppo del sistema ("*time-to-market*"), rendono la distribuzione dei sistemi ICT con pochi difetti un'attività estremamente impegnativa e complessa.

2.1.1 Verifica dell'Hardware e del Software

Le tecniche di verifica del sistema vengono applicate alla progettazione dei sistemi ICT in molti modi affidabili. In breve, viene utilizzata la verifica del sistema per stabilire se il progetto o il prodotto preso in considerazione possiede alcune proprietà. Le proprietà da convalidare possono essere abbastanza elementari, ad esempio, un sistema non dovrebbe mai essere in grado di raggiungere una situazione in cui non è possibile progredire (uno scenario di stallo o *deadlock*) e sono per lo più ottenute dalla *specifica di sistema*. Questa specifica prescrive cosa deve fare il sistema e cosa no, e costituisce quindi la base per qualsiasi attività di verifica. Un difetto viene trovato una volta che il sistema non soddisfa una delle proprietà della specifica. Il sistema è considerato essere "corretto" ogni volta che soddisfa tutte le proprietà ottenute dalle sue specifiche. In questo modo la correttezza è sempre relativa a una specifica e non è una proprietà assoluta di un sistema. Una vista schematica della verifica è illustrata nella Figura 2.2. Prima di introdurre la tecnica di verifica del modello (che parte da una specifica formale del sistema) e discutere del ruolo delle specifiche formali, vengono di seguito brevemente esaminate le tecniche di verifica alternative del software e dell'hardware.

Verifica del Software

La revisione tra pari (*peer review*) e il testing sono le principali tecniche di verifica del software utilizzate nella pratica. Una *revisione tra pari* equivale ad un'ispezione software effettuata da un team di ingegneri del software che preferibilmente non è stato coinvolto nello sviluppo del software in esame. Il codice non compilato non viene eseguito, ma analizzato completamente in modo statico. Studi empirici indicano che

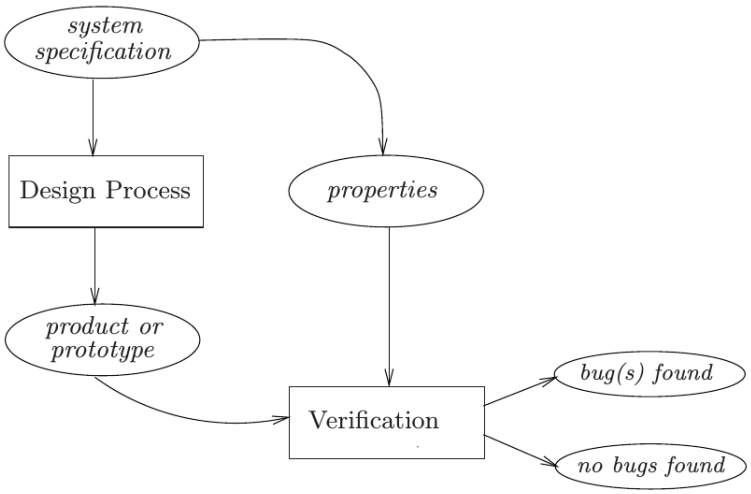

Figura 2.2: Vista schematica di una verifica di sistema a posteriori.

la revisione tra pari fornisce una tecnica efficace che cattura tra il 31% e il 93% dei difetti con una mediana intorno al 60%. Sebbene siano applicati per lo più in modo piuttosto ad hoc, esistono tipologie di procedure ancora più efficaci di revisione tra pari, ad esempio quelle focalizzate su specifici obiettivi di rilevamento degli errori. Nonostante sia quasi completamente manuale, la peer review è una tecnica piuttosto utile. Non sorprende quindi che una qualche forma di revisione tra pari venga utilizzata in quasi l'80% di tutti i progetti di ingegneria del software. Dovuto alla sua natura statica, l'esperienza ha dimostrato che errori insidiosi come errori riguardanti la concorrenza e gli algoritmi siano difficili da catturare usando la revisione tra pari. Il *testing* del software costituisce una parte significativa di qualsiasi progetto di ingegneria del software. Una parte compresa tra il 30% e il 50% dei costi totali di un progetto software sono dedicati ai test. Al contrario della revisione tra pari, che analizza il codice staticamente senza eseguirlo, il testing è una tecnica dinamica che esegue effettivamente il software. Il testing prende in considerazione una parte del software complessivo e fornisce il suo codice compilato con input, chiamati test. La correttezza è quindi ottenuta forzando il software a percorrere una serie di percorsi di esecuzione, sequenze di istruzioni di codice che rappresentano una esecuzione del software. Sulla base delle osservazioni fatte durante l'esecuzione del test, l'output effettivo del software viene confrontato con l'output previsto come documentato nella specifica di sistema. Sebbene la generazione di test e l'esecuzione di test possano essere parzialmente automatizzati, il confronto viene solitamente eseguito da persone. Il principale vantaggio del test è che può essere applicato a tutti i tipi di software, che vanno dal software applicativo (ad esempio il software per il commercio

elettronico) ai compilatori e ai sistemi operativi. Un test esaustivo di tutti i percorsi di esecuzione è praticamente impossibile; in pratica viene trattato solo un piccolo sottoinsieme di questi percorsi. Il testing può quindi non essere mai completo. Vale a dire, il test può mostrare solo la presenza di errori, non la loro assenza. Un altro problema con il test è quello di determinare quando fermarsi. In pratica, è difficile, se non impossibile, indicare l'intensità di test da effettuare per raggiungere un certo livello di densità dei difetti, ovvero il rapporto tra il numero di difetti e il numero di righe di codice non commentate. Gli studi hanno dimostrato che la revisione tra pari e il testing catturano differenti classi di difetti in diverse fasi del ciclo di sviluppo. Pertanto sono spesso usati insieme. Per aumentare l'affidabilità del software, questi approcci di verifica del software sono integrati con tecniche di miglioramento dei processi software, di progettazione strutturata e di metodi di specifica come ad esempio l'UML (*Linguaggio di modellazione Unificato*) e l'uso di sistemi di controllo di gestione della versione e della configurazione. Vengono utilizzate tecniche formali, in una forma o nell'altra, dal 10% al 15% circa di tutti i progetti software. *Catturare errori del software: prima è, meglio è.* È di grande importanza localizzare bug nel software. Lo slogan è: prima è, meglio è. I costi per riparare un difetto del software durante la manutenzione sono circa 500 volte più alti di una correzione fatta in una fase di progettazione iniziale (vedere la Figura 2.3). La verifica del sistema dovrebbe quindi aver luogo nella fase iniziale del processo di progettazione. Circa il 50% di

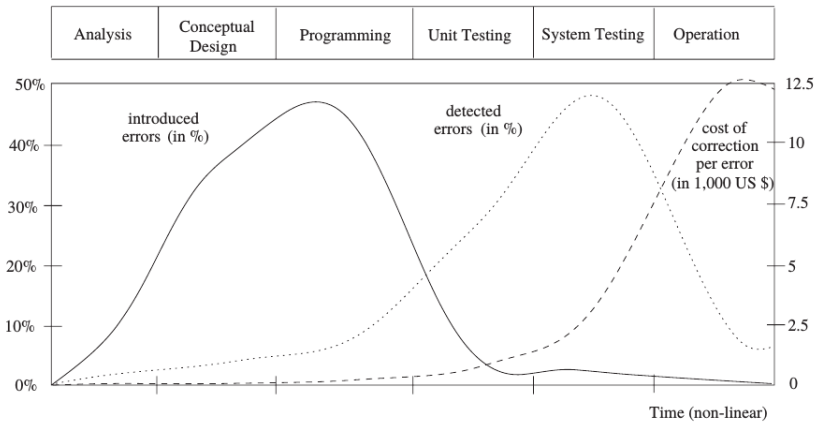

Figura 2.3: Ciclo di vita del software e costi di introduzione, di rilevamento e di riparazione di errori.

tutti i difetti viene introdotto durante la programmazione, la fase in cui la scrittura effettiva del codice ha luogo. Mentre solo il 15% di tutti gli errori viene rilevato nelle fasi iniziali di progettazione, la maggior parte degli errori viene rilevato durante il testing. All'inizio del *test di unità*, che è orientato a scoprire i difetti nei singoli mo-

duli software che compongono il sistema, una densità di difetti di circa 20 difetti per 1000 righe di codice (non commentato) è tipica. Questo valore è stato ridotto a circa 6 difetti per 1000 righe di codice all'inizio del test del sistema, dove viene testata una collezione di moduli che costituisce il prodotto finale. Al momento del lancio di un nuovo rilascio del software, la tipica densità di difetti del software accettata è di circa un difetto per 1000 linee di codice. Gli errori sono in genere concentrati in alcuni moduli software, circa la metà dei moduli sono privi di difetti e circa l'80% dei difetti si presenta in una piccola percentuale (circa il 20%) dei moduli, e spesso gli errori si verificano quando si interfacciano i moduli. La riparazione degli errori che sono rilevati prima del test può essere ottenuta in maniera economica. Il costodella riparazione aumenta significativamente da circa 1000 USD (per riparazione di un errore) durante i test di unità ad un massimo di circa 12,500 USD quando il difetto viene riscontrato solo durante il funzionamento del sistema. È di vitale importanza ricercare tecniche che individuino i difetti il prima possibile nella progettazione del processo di progettazione del software: i costi per ripararli sono sostanzialmente inferiori e la loro influenza sul resto del progetto è meno sostanziale.

Verifica dell'Hardware

La prevenzione degli errori nella progettazione dell'hardware è di vitale importanza. L'hardware è soggetto ad alti costi di fabbricazione; risolvere i difetti dopo la consegna ai clienti è difficile, e le aspettative di qualità sono alte. Considerando che i difetti del software possono essere riparati fornendo patch o aggiornamenti – al giorno d'oggi gli utenti tendono persino ad anticipare e ad accettare questo – le correzioni di errori dell' hardware dopo la consegna ai clienti sono molto difficili e richiedono principalmente una nuova fabbricazione e relativa redistribuzione. Questo ha conseguenze economiche enormi. Il rimpiazzo dei processori Pentium II difettosi ha causato ad Intel una perdita di circa 475 milioni di dollari. La legge di Moore – il numero di porte logiche in un circuito raddoppia ogni 18 mesi – ha dimostrato di essere vera nella pratica ed è un grosso ostacolo alla produzione di hardware corretto. Studi empirici hanno indicato che oltre il 50% di tutti gli ASIC (Circuiti Integrati per Applicazioni Specifiche) non funzionano correttamente dopo la progettazione iniziale e la fabbricazione. Non è sorprendente che i produttori di microchip investano molto per ottenere progetti corretti. La verifica dell'hardware è una parte ben consolidata del processo di progettazione. Lo sforzo progettuale in un tipico progetto hardware rappresenta solo il 27% del tempo totale speso per il chip; il resto del tempo è dedicato al rilevamento e alla prevenzione degli errori.

Tecniche di verifica dell'hardware. Emulazione, simulazione e analisi strutturale sono le tecniche principali utilizzate nella verifica dell'hardware.

L'analisi strutturale comprende diverse tecniche specifiche come sintesi, analisi del tempo, e controllo di equivalenza che non sono descritti in questo lavoro.

L'*emulazione* è una sorta di test. Un sistema hardware generico riconfigurabile (l'emulatore) viene configurato in modo tale che si comporti come il circuito considerato e quindi viene ampiamente testato. Come nel caso del test del software, l'emulazione equivale a fornire una serie di stimoli al circuito confrontando l'uscita generata con l'uscita prevista come indicato nella specifica del chip. Per testare completamente il circuito, tutte le possibili combinazioni di input in ogni possibile stato del sistema dovrebbe essere esaminate. Questo è poco pratico e il numero di test deve essere ridotto in modo significativo, generando nuovi potenziali errori.

Con la *simulazione* , un modello del circuito in questione viene costruito e simulato. I modelli sono in genere forniti utilizzando linguaggi di descrizione dell'hardware come Verilog o VHDL, entrambi standardizzati da IEEE. In base agli stimoli, i percorsi di esecuzione del modello del chip sono esaminati utilizzando un simulatore. Questi stimoli possono essere forniti da un utente o automatizzati tramite un generatore casuale. Una mancata corrispondenza tra l'uscita del simulatore e l'uscita descritta nella specifica determina la presenza di errori. La simulazione è come il test, ma viene applicato ai modelli. Tuttavia, soffre degli stessi limiti: il numero di scenari da verificare in un modello per ottenere piena fiducia va oltre ogni ragionevole sottoinsieme di scenari che possono essere esaminati nella pratica.

La simulazione è la tecnica di verifica dell'hardware più popolare e viene utilizzata in varie fasi di progettazione, ad esempio a livello di trasferimento del registro, livello di gate e di transistor. Oltre a queste tecniche di rilevamento di errori, il *testing dell'hardware* è necessario per trovare difetti di fabbricazione derivanti da difetti di configurazione nel processo di fabbricazione.

2.2 Verifica del Modello (Model Checking)

Nella progettazione software e hardware di sistemi complessi, si impiegano più tempo e sforzi per la verifica piuttosto che per la costruzione. Si cercano tecniche per ridurre e facilitare gli sforzi per la verifica aumentando al contempo la loro copertura. I metodi formali offrono un grande potenziale per ottenere una integrazione precoce della verifica nel processo di progettazione, per fornire tecniche di verifica più efficaci e ridurre i tempi di verifica.

Vediamo prima brevemente il ruolo dei metodi formali. Per dirlo in poche parole, i metodi formali possono essere considerati come "la matematica applicata per la modellazione e l'analisi dei sistemi ICT". Il loro scopo è quello di stabilire la correttezza del sistema con rigore matematico. Il loro grande potenziale ha portato ad un uso crescente da parte degli ingegneri di metodi formali per la verifica di sistemi complessi software ed hardware. Inoltre, i metodi formali sono una delle tecniche di verifica "altamente raccomandate" per lo sviluppo di software di sistemi di sicurezza e di sistemi critici secondo, ad esempio, le migliori pratiche standard della IEC (Commissione Internazionale di Elettrotecnica) e norme dell'ESA (Agenzia Spaziale Europea). Il rapporto risultante di un'indagine da parte della FAA (Autorità Federale dell'Aviazione) e della NASA (Amministrazione Nazionale dell'Aeronautica e dello Spazio) sull'uso di metodi formali conclude che:
I metodi formali dovrebbero far parte dell'educazione di ogni scienziato informatico e ingegnere del software, proprio come il ramo appropriato della matematica applicata è un parte necessaria dell'educazione di tutti gli altri ingegneri.

Durante gli ultimi due decenni, la ricerca sui metodi formali ha portato allo sviluppo di alcune tecniche di verifica molto promettenti che facilitano il rilevamento precoce dei difetti. Queste tecniche sono accompagnate da potenti strumenti software che possono essere utilizzati per automatizzare varie fasi della verifica. Le indagini hanno dimostrato che le procedure di verifica formale avrebbe rivelato i difetti esposti , ad esempio, nel missile Ariane-5, nel Mars Pathfinder, nel Processore Intel Pentium II e nella macchina per terapie Therac-25.

Le tecniche di verifica *basate sul modello* si basano su modelli che descrivono il possibile comportamento del sistema in modo matematicamente preciso e non ambiguo. Si scopre che – prima di qualsiasi forma di verifica – la modellazione accurata dei sistemi spesso porta alla scoperta di incompletezza, ambiguità e incongruenza nelle specifiche informali del sistema. Solitamente tali problemi vengono scoperti solo in una fase successiva del progetto. I modelli del sistema sono corredati da algoritmi che esplorano sistematicamente tutti gli stati del modello del sistema. Questo fornisce la base per tutta una serie di tecniche di verifica che vanno da una esplorazione esaustiva (model checking) ad esperimenti con un insieme restrittivo di scenari nel modello (simulazione) o nella realtà (test). A causa dei miglioramenti incessanti degli algoritmi e delle strutture dati sottostanti, insieme alla disponibilità di computer più veloci e con memorie sempre più capienti, le tecniche basate su modelli che solo un decennio fa hanno funzionato per esempi molto semplici sono oggigiorno applicabili a progetti realistici. Poiché il punto di partenza di queste tecniche è un modello del sistema preso in considerazione, abbiamo come dato di fatto che

Qualsiasi verifica utilizzando tecniche basate su modelli è valida solo quanto è valido il modello del sistema.

La verifica del modello è una tecnica di verifica che esplora tutti i possibili stati del sistema in maniera forza-bruta. Simile ad un programma per il gioco degli scacchi per computer che controlla le mosse possibili, un model checker, lo strumento software che esegue il model checking, esamina tutti i possibili scenari di sistema in modo sistematico. In questo modo, può essere mostrato che un dato modello di sistema soddisfa veramente una certa proprietà. È una vera sfida esaminare il più grande possibile spazio degli stati che possono essere trattati con mezzi attuali, cioè processori e memorie. I model checkeri allo stato dell'arte possono gestire spazi di circa da 10^8 a 10^9 stati con esplicita enumerazione dello spazio degli stati. Utilizzando algoritmi intelligenti e strutture dati personalizzate, possono essere gestiti per problemi specifici spazi di stato più grandi(da 10^{20} fino ad anche 10^{476} stati) . Persino gli errori insidiosi che non vengono scoperti usando l'emulazione, il test e la simulazione possono essere rivelati utilizzando il model checking.

Le proprietà tipiche che possono essere verificate utilizzando il model checking sono di natura qualitativa: Il risultato generato è OK?, Il sistema può raggiungere una situazione di stallo, ad esempio, quando due programmi concorrenti si stanno aspettando l'un l'altro e quindi bloccano l'intero sistema? Ma anche le proprietà di temporizzazione possono essere controllate: può verificarsi una situazione di stallo entro 1 ora dopo un reset di sistema?, oppure, la risposta è sempre ricevuta entro 8 minuti? Il model checking richiede una affermazione precisa e inequivocabile delle proprietà da esaminare. Come nel fare un accurato modello di sistema, questo passaggio porta spesso alla scoperta di diverse ambiguità e incongruenze nella documentazione informale. Ad esempio, la formalizzazione di tutte le proprietà di sistema per un sottoinsieme della parte utente del protocollo ISDN hanno rivelato che il 55% (!) dei requisiti originali e informali di sistema erano incoerenti.

Generalmente il modello del sistema viene generato automaticamente da una descrizione del modello specificato in un qualche dialetto appropriato di linguaggi di programmazione come C o Java o linguaggi di descrizione dell'hardware come Verilog o VHDL. Si noti che la specifica della proprietà prescrive cosa deve fare il sistema e cosa non dovrebbe fare, mentre la descrizione del modello affronta il comportamento del sistema. Il model checker esamina tutti i pertinenti stati del sistema per verificare se soddisfano la proprietà desiderata. Se si incontra uno stato che viola la proprietà in esame, il model checker fornisce un controesempio che indica come il modello potrebbe raggiungere lo stato indesiderato. Il controesempio descrive un percorso di esecuzione che conduce dallo stato iniziale del sistema ad uno stato che viola la

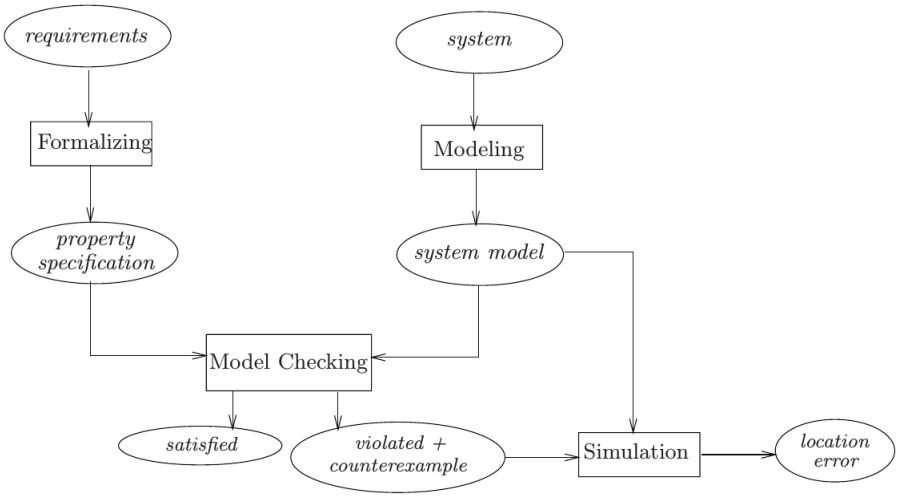

Figura 2.4: Vista schematica dell'approccio della verifica del modello.

proprietà che viene verificata. Con l'aiuto di un simulatore, l'utente può riprodurre lo scenario che contiene la violazione, in modo da ottenere utili informazioni di debug e adattare il modello (o la proprietà) di conseguenza (si veda la Figura 2.4).

Il model checking è stato applicato con successo a diversi sistemi ICT e alle loro applicazioni. Ad esempio, sono stati rilevate delle situazioni di stallo nei sistemi di prenotazione online delle compagnie aeree, i protocolli di commercio elettronico sono stati verificati e sono stati apportati miglioramenti significativi delle specifiche del sistema sugli standard internazionali IEEE per la comunicazione interna degli elettrodomestici. Cinque errori precedentemente non scoperti sono stati identificati in un modulo di esecuzione del controllo del veicolo spaziale Deep Space 1, in un caso identificando un grande difetto di progettazione. Un bug identico a quello scoperto dal model checking sfuggito al testing e che ha causato una situazione di stallo durante un esperimento di volo a 96 milioni di km dalla Terra. Nei Paesi Bassi, il model checking ha rivelato diversi seri difetti di progettazione nel software di controllo di una barriera anti-tempesta che protegge il porto principale di Rotterdam dalle inondazioni.

2.3 Caratteristiche del Model Checking

Partendo dal principio del Model Checking:
Il model checking è una tecnica automatica che, dato un modello a stati finiti di un sistema e una proprietà formale, controlla sistematicamente se questa proprietà è valida per (un dato stato in) quel modello.

descriviamo nelle seguenti sezioni i dettagli tecnici elementari del model checking, descrivendo il processo (come usarlo), presentando i suoi principali vantaggi e inconvenienti e discutendo il suo ruolo nel ciclo di sviluppo del sistema.

2.3.1 Il processo di verifica del modello

Nell'applicare il model checking ad un progetto si possono distinguere le seguenti diverse fasi:

- Fase di *modellazione*:
  - modellare il sistema in esame utilizzando il linguaggio di descrizione del model checker disponibile;
  - come primo controllo di integrità e valutazione rapida del modello eseguire alcune simulazioni;
  - formalizzare la proprietà da verificare utilizzando il linguaggio delle specifiche della proprietà.
- Fase di *esecuzione*: eseguire il model checker per verificare la validità della proprietà nel modello del sistema.
- Fase di *analisi*:
  - la proprietà è soddisfatta? → controllare la proprietà successiva (se presente);
  - la proprietà è violata? →
    1. analizzare il controesempio generato mediante simulazione;
    2. perfezionare il modello, il progetto o la proprietà;
    3. ripetere l'intera procedura.
  - Memoria insufficiente? → provare a ridurre il modello e riprovare.

Oltre a questi passaggi, l'intera verifica dovrebbe essere pianificata, amministrata e organizzata. Questo procedimento è chiamato *organizzazione della verifica*. Discutiamo queste fasi in modo più dettagliato di seguito.

Modellazione

I prerequisiti per il model checking sono un modello del sistema sotto considerazione e una caratterizzazione formale della proprietà da controllare.

I modelli di sistemi descrivono il comportamento dei sistemi in modo preciso e non ambiguo. Sono per lo più espressi usando automi a stati finiti, costituiti da un insieme finito di stati e un insieme di transizioni. Gli stati contengono informazioni sui valori correnti delle variabili, l'istruzione precedentemente eseguita (ad esempio un program counter) e cose simili. Le transizioni descrivono come il sistema si evolve da uno stato all'altro. Per i sistemi realistici, gli automi a stati finiti sono descritti usando un linguaggio di descrizione del modello come ad esempio un appropriato dialetto/estensione di C, Java, VHDL o simili. Modellare i sistemi, in particolare quelli concorrenti, ad un livello di astrazione adeguato è piuttosto complicato ed è davvero un arte.

Al fine di migliorare la qualità del modello, può avere luogo una simulazione prima del model checking. La simulazione può essere utilizzata efficacemente per sbarazzarsi della categoria più semplice degli errori di modellazione. Eliminare gli errori più semplici prima che qualsiasi controllo approfondito abbia luogo, potrebbe ridurre il costo e lo sforzo in termini di tempo per la verifica.

Per rendere possibile una verifica rigorosa, le proprietà dovrebbero essere descritte in modo preciso e in maniera non ambigua. Questo viene tipicamente eseguito usando un linguaggio di specifica di proprietà. Ci concentriamo in particolare sull'uso di una *logica temporale* come linguaggio di specifica delle proprietà, una forma di logica modale che è appropriata per specificare le proprietà rilevanti dei sistemi ICT. In termini di logica matematica, si controlla che la descrizione del sistema sia un modello di una formula in logica temporale. Questo spiega il termine "*model checking*". La logica temporale è fondamentalmente un'estensione della logica proposizionale tradizionale con operatori che si riferiscono al comportamento dei sistemi nel tempo. Permette la specifica di una vasta gamma di proprietà pertinenti del sistema come la correttezza funzionale (il sistema fa ciò che è supposto fare?), raggiungibilità (è possibile finire in uno stato di stallo?), sicurezza ("qualcosa di brutto non succede mai"), *liveness* ("qualcosa di buono alla fine accadrà"), equità (in determinate condizioni, un evento si verifica ripetutamente?) e le proprietà in tempo reale (il sistema agisce in tempo?).

Sebbene i passaggi sopra menzionati siano spesso ben compresi, in pratica potrebbe essere un problema serio giudicare se l'affermazione del problema formalizzato (modello + proprietà) sia una descrizione adeguata del vero problema di verifica. Questo è anche conosciuto come il *problema di validazione*. La complessità del sistema coinvolto, così come la mancanza di precisione delle specifiche informali della funzionalità del sistema, potrebbe rendere difficile rispondere a questa domanda in modo soddisfacente. La verifica e la convalida non devono essere confuse. La verifica

equivale a verificare che il progetto soddisfi i requisiti che sono stati identificati, vale a dire, la verifica è "controllare che stiamo costruendo la cosa giusta". In fase di validazione, viene verificato se il modello formale è coerente con la concezione informale del progetto, cioè la convalida è "controllare che stiamo verificando la cosa giusta".

Esecuzione del Model Checker

Il Model Checker deve prima essere inizializzato impostando opportunamente le varie opzioni e direttive che possono essere utilizzate per eseguire la verifica esaustiva. Successivamente, viene effettuato il model checking. Questo è fondamentalmente un approccio esclusivamente algoritmico in cui la validità della proprietà considerata viene verificata in tutti gli stati del modello di sistema.

Analisi dei risultati

Ci sono fondamentalmente tre risultati possibili: la proprietà specificata è valida nel modello oppure no, oppure il modello risulta troppo grande per adattarsi ai limiti fisici della memoria del computer.

Nel caso in cui la proprietà sia valida, è possibile controllare la proprietà seguente o, nel caso in cui tutte le proprietà sono stati controllate, si conclude che il modello possiede tutte le proprietà desiderate.

Ogni volta che una proprietà viene falsificata, il risultato negativo può avere cause diverse. Potrebbe essere un errore di modellazione , cioè studiando l'errore si scopre che il modello non riflette il progetto del sistema. Ciò implica una correzione del modello e la verifica deve essere riavviata con il modello migliorato. Questo nuovo riavvio include la verifica di quelle proprietà che sono state controllate in precedenza sul modello errato e la cui verifica potrebbe essere invalidata dalla correzione del modello! Se l'analisi dell'errore mostra che non c'è indebita discrepanza tra la progettazione e il suo modello, quindi è presente un errore di progettazione o si è verificato un errore d proprietà. In caso di errore di progettazione, la verifica si conclude con un risultato negativo, e il progetto (insieme al suo modello) deve essere migliorato. Può essere il caso che dopo aver studiato l'errore che si è rivelato, si scopre che la proprietà non riflette il requisito informale che doveva essere convalidato. Ciò implica una modifica della proprietà, e una nuova verifica del modello deve essere effettuata. Poiché il modello non viene modificato, non è necessario effettuare alcuna riverifica delle proprietà precedentemente verificate. Il progetto è verificato se e solo se tutte le proprietà sono state verificate rispetto ad un modello valido.

Ogni volta che il modello è troppo grande per essere gestito – gli spazi degli stati dei sistemi utilizzati nella vita reale possono essere di molti ordini di grandezza più

grandi di quelli che possono essere memorizzati dalle memorie attualmente disponibili – ci sono vari modi per procedere. Una possibilità è quella di applicare tecniche che cercano di sfruttare le regolarità implicite nella struttura del modello. Esempi di queste tecniche sono le rappresentazioni degli spazi di stato usando tecniche simboliche come i diagrammi di decisione binaria o riduzione dell'ordine parziale. In alternativa vengono utilizzate astrazioni rigorose del modello del sistema completo. Queste astrazioni dovrebbero preservare la (non) validità delle proprietà che deve essere controllato. Spesso si possono ottenere astrazioni sufficientemente piccole rispetto a una singola proprietà. In tal caso, è necessario creare diverse astrazioni del modello a disposizione. Un altro modo di gestire gli spazi di stato troppo grandi consiste nel rinunciare alla precisione del risultato della verifica. Gli approcci di verifica probabilistica esplorano solo una parte dello spazio degli stati mentre si effettua un sacrificio (spesso trascurabile) nella copertura della verifica.

Organizzazione della verifica

L'intero processo di verifica del modello dovrebbe essere ben organizzato, ben strutturato e ben pianificato. Applicazioni industriali di model checking hanno evidenziato che l'uso della versione e della gestione della configurazione è di particolare rilevanza. Durante la procedura di verifica, ad esempio, sono disponibili diverse descrizioni dei modelli realizzati che descrivono diverse parti del sistema, varie versioni del modello di verifica (ad es. a causa dell'astrazione) e inoltre sono disponibili molti parametri di verifica (es. opzioni di controllo del modello) e risultati (tracce diagnostiche, statistiche). Questa informazione deve essere documentata e mantenuta con molta attenzione per poter gestire un processo pratico di model checking e per consentire la riproduzione degli esperimenti che sono stati effettuati.

2.3.2 Punti di forza e debolezza

I punti di forza del model checking:

- È un approccio di verifica *generale* applicabile ad un'ampia gamma di applicazioni come sistemi integrati, ingegneria del software e progettazione di hardware.
- Supporta la verifica *parziale*, cioè le proprietà possono essere controllate individualmente, quindi permettendo prima di focalizzare l'attenzione sulle proprietà essenziali. Non à necessaria alcuna specifica del requisito completo.

- Non è vulnerabile alla probabilità che venga rilevato un errore; questo contrasta con il testing e la simulazione che sono mirati a rintracciare i difetti più probabili.
- Fornisce *informazioni diagnostiche* nel caso in cui una proprietà venga invalidata; questo è molto utile per scopi di debug.
- È potenzialmente una tecnologia "*push-button*"; l'uso del model checking non richiede né un alto grado di interazione con l'utente e nemmeno un alto livello di esperienza.
- Gode di un *interesse* in rapido aumento da parte dell'*industria*; diverse società di hardware hanno avviato i loro laboratori di verifica interni, appaiono frequentemente offerte di lavoro con competenze richieste in model checking, e model checker commerciali sono diventati disponibili.
- Può essere facilmente *integrato* nei cicli di sviluppo esistenti; la sua curva di apprendimento non è molto ripida e studi empirici indicano che potrebbe portare a tempi di sviluppo più brevi.
- Ha una *base solida e matematica*; si basa sulla teoria degli algoritmi su grafo, delle strutture dati e della logica.

I punti deboli del model checking:

- È particolarmente appropriato per le applicazioni ad *alta intensità di controllo* e meno adatto per applicazioni ad alta intensità di dati in quanto i dati in genere si estendono su domini infiniti.
- La sua applicabilità è soggetta a *problemi di decidibilità*; per sistemi a stati infiniti, o per ragionamento su tipi di dati astratti (che richiedono logiche indecidibili o semi-decidibili), il model checking non è in genere efficacemente calcolabile.
- Verifica un *modello di sistema* e non il sistema reale (prodotto o prototipo); qualsiasi risultato ottenuto è quindi buono come è buono il modello di sistema. Tecniche complementari come il testing, sono necessarie per trovare difetti di fabbricazione (per l'hardware) o errori nel codice (per il software).
- Controlla solo i *requisiti dichiarati*, cioè non c'è garanzia di completezza. La validità delle proprietà che non sono controllate non può essere giudicata.

- Soffre del problema dell'*esplosione dello spazio degli stati*, cioè il numero di stati necessari per modellare il sistema con precisione può facilmente superare la quantità di memoria disponibile del computer. Nonostante lo sviluppo di diversi metodi molto efficaci per combattere questo problema, i modelli di sistemi realistici potrebbero essere ancora troppo grandi per adattarsi alla memoria.

- Il suo utilizzo richiede una certa *esperienza* nella ricerca di astrazioni appropriate per ottenere modelli di sistema di dimensioni minori e per dichiarare le proprietà nel formalismo logico utilizzato.

- Non è garantito il raggiungimento di risultati corretti: come con qualsiasi strumento, un model checker può contenere *difetti del software*.

- Non consente la verifica delle *generalizzazioni*: in generale, i sistemi di controllo con un numero arbitrario di componenti, o i sistemi parametrizzati, non possono essere trattati. Il model checking può, tuttavia, suggerire risultati per parametri arbitrari che possono essere verificati usando gli *assistenti di dimostrazione*.

Crediamo che non si possa mai raggiungere la correttezza assoluta garantita per i sistemi di dimensione realistica. Nonostante le limitazioni di cui sopra, possiamo concludere che
Il model checking è una tecnica efficace per rivelare potenziali errori di progettazione.
Pertanto, il model checking può fornire un significativo aumento del livello di confidenza del progetto di un sistema.

2.4 Logica Temporale Lineare (LTL)

2.4.1 Sintassi e semantica LTL.

Le formule della Logica Temporale Lineare (LTL) (vedi Pnueli (1977)) sono costruite su un insieme finito di proposizioni atomiche come segue:

$$\phi = p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathcal{X}\,\phi_1 \mid \phi_1\,\mathcal{U}\,\phi_2 \mid (\phi)$$

dove $p \in Prop$, ϕ, ϕ_1, ϕ_2 sono formule LTL, $\mathcal{X}$ è l'operatore "*next*" e $\mathcal{U}$ è l'operatore "*until*". Di seguito, se non diversamente specificato attraverso l'utilizzo di parentesi, gli operatori unari hanno una precedenza più alta rispetto agli operatori binari. Una formula LTL è interpretata attraverso una *computazione*, cioè una funzione $\pi : \mathbb{N} \to 2^{Prop}$ che assegna valori di verità agli elementi di $Prop$ ad ogni istante (numero naturale). Per una computazione π e un istante di tempo $i \in \mathbb{N}$:

- $\pi, i \models p$ per $p \in Prop$ se e solo se $p \in \pi(i)$
- $\pi, i \models \neg\alpha$ se e solo se $\pi, i \not\models \alpha$
- $\pi, i \models (\alpha \vee \beta)$ se e solo se $\pi, i \models \alpha$ or $\pi, i \models \beta$
- $\pi, i \models \mathcal{X}\,\alpha$ se e solo se $\pi, i+1 \models \alpha$
- $\pi, i \models \alpha\,\mathcal{U}\,\beta$ se e solo se per alcuni $j \geq i$, abbiamo $\pi, j \models \beta$ e per tutti i k, $i \leq k < j$ abbiamo $\pi, k \models \alpha$

Diciamo che π *soddisfa* una formula ϕ, denotata con $\pi \models \phi$, se e solo se $\pi, 0 \models \phi$. Se $\pi \models \phi$ per ogni π, allora ϕ è *vera* e scriviamo $\models \phi$.

Consideriamo altri connettivi booleani come "$\wedge$" e "$\rightarrow$" con il significato usuale, e abbreviamo $p \vee \neg p$ con $\top$, $p \wedge \neg p$ con $\bot$. Introduciamo $\Diamond\,\phi$ ("*eventually*") per denotare $\top\,\mathcal{U}\,\phi$ e $\Box\,\phi$ ("*always*") per denotare $\neg\Diamond\,\neg\phi$. Infine, alcuni PSP usano l'operatore "*weak until*" definito come $\alpha\,\mathcal{W}\,\beta = \Box\,\alpha \vee (\alpha\,\mathcal{U}\,\beta)$.

2.4.2 Soddisfacibilità LTL.

Tra i diversi approcci per decidere la soddisfacibilità del linguaggio LTL, quello verso il model checking è stato proposto in Rozier and Vardi (2007) per verificare la consistenza dei requisiti espressi come formule LTL. Data una formula ϕ su un set $Prop$ di proposizioni atomiche, può essere costruito un modello *universale* M. Intuitivamente, un modello universale codifica tutti le computazioni possibili su $Prop$ come tracce (infinite), e quindi ϕ è soddisfacibile precisamente quando M non soddisfa $\neg\phi$. In Rozier and Vardi (2011) viene presentato un primo miglioramento rispetto a questa strategia di base insieme allo strumento PANDA mentre in Li et al. (2013b) viene proposto un algoritmo basato sulla costruzione di automi per migliorare ulteriormente le prestazioni — l'approccio è implementato in uno strumento chiamato AALTA. Ulteriori studi lungo questa direzione includono Li et al. (2014) e Li et al. (2013a). Nel secondo strumento, viene proposto di eseguire un portfolio di risolutori di soddisfacibilità LTL chiamato POLSAT in modo da eseguire diverse tecniche in parallelo e restituire il risultato del primo che termina con successo.

2.5 Verifica della soddisfacibilità per LTL

Il problema della soddisfacibilità per LTL è: per una determinata formula LTL φ esiste un modello per quale φ è verificata? Cioè, abbiamo $Words(\varphi) \neq \emptyset$? La soddisfacibilità può essere risolta costruendo un NBA (Automa di Büchi Nondeterministico) $\mathcal{A}_\varphi$ per la formula LTL φ . In questo modo, l'esistenza di una parola

infinita $\sigma \in \mathit{Words}(\varphi) = \mathcal{L}_\omega(\mathcal{A}_\varphi)$ può essere provata. Il problema della vacuità per il NBA $\mathcal{A}$, ovvero se $\mathcal{L}_\omega(\mathcal{A}) = \emptyset$ oppure no, può essere risolto mediante una tecnica simile alla verifica della persistenza, come mostrato nell'Algoritmo 2.1. Oltre a una risposta affermativa, un prefisso di una parola $\sigma \in \mathcal{L}_\omega(\mathcal{A}) = \mathit{Words}(\varphi)$ può essere fornito in modo simile ad un controesempio per il model checking LTL.

Algoritmo 2.1 Verifica della soddisfacibilità LTL

Input: Formula LTL φ su AP

Output: "*si*" se la formula φ è soddisfacibile. Altrimenti "*no*".

Costruire un NBA $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ con $\mathcal{L}_\omega(\mathcal{A}) = Words(\varphi)$

(* Verificare se $\mathcal{L}_\omega(\mathcal{A}) = \emptyset$. *)

Eseguire una Ricerca in Profondità (DFS) nidificata per determinare se esiste uno stato $q \in F$ raggiungibile da $q_0 \in Q_0$ e che si trovi su un ciclo

Se è così allora ritornare "*si*". Altrimenti "*no*".

Dato che la soddisfacibilità per LTL può essere affrontata utilizzando tecniche simili a quelle del model checking LTL, consideriamo ora il problema della validità. La formula φ è valida ogni volta che φ è valida sotto tutte le interpretazioni, cioè, $\varphi \equiv \mathit{true}$.
Per la formula LTL φ su AP abbiamo che φ è valida se e solo se $\mathit{Words}(\varphi) = (2^{AP})^\omega$. La validità di φ può essere stabilita osservando che φ è valida se e solo se $\neg\varphi$ non è soddisfacente. Quindi, in modo algoritmico controlla se si ottiene φ, si costruisce un NBA per $\neg\varphi$ e si applica l'algoritmo di soddisfacibilità (vedi Algoritmo 2.1) a $\neg\varphi$.

L'algoritmo di soddisfacibilità LTL descritto ha un tempo di esecuzione esponenziale nella lunghezza di φ. Il seguente risultato mostra che una tecnica essenzialmente più efficiente non può essere ottenuta poichè sia i problemi di validità che di soddisfacibilità sono PSPACE-hard. In effetti, entrambi i problemi sono anche PSPACE-completi. L'appartenenza a PSPACE per il problema della soddisfacibilità LTL può essere dimostrata fornendo un algoritmo non deterministico polinomialmente a spazio-limitato che indovini una esecuzione finita nel GNBA (Automa di Büchi Nondeterministico Generalizzato) $\mathcal{G}_\varphi$ per la formula φ e controlla se questa esecuzione finita è un prefisso di una esecuzione accettata in $\mathcal{G}_\varphi$. Il fatto che il problema della validità LTL appartiene a PSPACE può essere derivato dall'osservazioni

Response

Descrive le relazioni causa-effetto tra una coppia di eventi/stati. Un'occorrenza del primo, la causa, deve essere seguita da un'occorrenza del secondo, l'effetto. Conosciuto anche come *Follows* e *Leads-to*.

Grammatica Inglese Strutturata
It is always the case that if P holds, then S eventually holds.

Mapping LTL

Globally	$\square\,(P \rightarrow \Diamond\, S)$
Before R	$\Diamond\, R \rightarrow (P \rightarrow (\overline{R}\ \mathcal{U}\ \ (S \wedge \overline{R})))\ \mathcal{U}\ \ R$
After Q	$\square\,(Q \rightarrow \square\,(P \rightarrow \Diamond\, S))$
Between Q and R	$\square\,((Q \wedge \overline{R} \wedge \Diamond\, R) \rightarrow (P \rightarrow (\overline{R}\ \mathcal{U}\ \ (S \wedge \overline{R})))\ \mathcal{U}\ \ R)$
After Q until R	$\square\,(Q \wedge \overline{R} \rightarrow ((P \rightarrow (\overline{R}\ \mathcal{U}\ \ (S \wedge \overline{R})))\ \mathcal{W}\ \ R)$

Example
It is always the case that if object_detected holds , *then* moving_to_target *eventually holds.*

Figura 2.5: Modello *Response* ($\overline{\alpha}$ sta per $\neg\alpha$).

che φ è valido se e solo se $\neg\varphi$ non è soddisfacibile e che PSPACE è chiuso sotto complementazione.

2.6 Modelli di specifica della proprietà (PSP)

La proposta originale dei PSP si trova in Dwyer et al. (1999). Sono pensati per descrivere la struttura dei comportamenti dei sistemi e fornire espressioni di tali comportamenti in una gamma di formalismi comuni. Un esempio di PSP è dato in Figura 2.5 - con alcune parti omesse per questioni di leggibilità. Un modello è composto da un *Nome* (Response nella Figura 2.5), una dichiarazione (informale) che descrive il comportamento assunto dal modello e una dichiarazione (Inglese strutturato) (vedi Konrad and Cheng (2005)) che dovrebbe essere usata per esprimere i requisiti. Vengono inoltre forniti i mapping LTL corrispondenti alle diverse declinazioni del modello, dove le lettere maiuscole (P, S, T, ecc.) rappresentano stati/eventi booleani. Più in dettaglio, un PSP è composto da due parti: (i) l'ambito (*scope*) e (ii) il corpo (*body*). Lo *scope* è la misura dell'esecuzione del programma su cui deve essere mantenuto il PSP, e ci sono cinque ambiti consentiti: *Globally*, per coprire

l'intera esecuzione dello scope; *Before*, per coprire l'esecuzione fino a uno stato/evento; *After*, per coprire l'esecuzione dopo uno stato/evento; *Between*, per coprire la parte di esecuzione da uno stato/evento ad un altro; *After-until*, dove la prima parte del pattern continua anche se il secondo stato/evento non accade mai. Per gli scope delimitati dallo stato, l'intervallo in cui la proprietà viene valutata è chiusa a sinistra ed aperta a destra. Il *body* di un modello descrive il comportamento che vogliamo specificare. In Dwyer et al. (1999), i body sono classificati in modelli di *occorrenza* e di *ordine*. I modelli di *occorrenza* richiedono che gli stati/eventi si verifichino o non si verifichino. Esempi di tali body sono *Absence* (Assenza), in cui un determinato stato/evento non deve verificarsi all'interno di uno scope, e il suo opposto *Existence* (Esistenza). I modelli di *ordine* vincolano l'ordine degli stati/eventi. Esempi di tali modelli sono *Precedence* (Precedenza), dove uno stato/evento deve sempre precedere un altro stato/evento e *Response* (Risposta), in cui uno stato/evento deve sempre essere seguito da un altro stato/evento all'interno dello scope. Inoltre, abbiamo incluso il modello *Invariant* introdotto in Post and Hoenicke (2012) e che stabilisce che uno stato/evento deve verificarsi ogni volta che si verifica un altro stato/evento. Combinando scope e body si possono costruire 55 diversi tipi di modelli.

2.6.1 Verifica di inconsistenza.

Di solito, l'inconsistenza in un insieme di requisiti viene spiegata in termini di sottoinsiemi minimi di requisiti che espongono i problemi principali all'interno della specifica. La letteratura non fornisce una denominazione coerente di tali nuclei, e il sottoinsieme minimo di incoerenza dei termini di *Minimo Sottoinsieme Inconsistente* (*MIS*) (vedi Bendík (2017)), *Minimo Sottoinsieme Insoddisfacibile* (*MUS*) (vedi Belov and Marques-Silva (2012)), *Minimo Nucleo Insoddisfacibile* (*MUC*) (vedi Liffiton and Sakallah (2008)), e anche *MUC di alto livello* (HLMUC) (vedi Nadel (2010)) sono stati introdotti per riferirsi allo stesso concetto – nel seguito e in tutto il documento, denotiamo con MUC un insieme minimo di requisiti inconsistenti. Gli algoritmi per trovare i MUC possono essere divisi in due gruppi di base: (*i*) quelli che si concentrano sull'estrazione di un singolo MUC e (*ii*) quelli che si concentrano sull'estrazione di tutti i MUC. Queste tecniche possono essere ulteriormente divise in domini specifici, vale a dire, mirare a domini specifici come la soddisfacibilità proposizionale (vedi Belov and Marques-Silva (2011)) , e per uso generale, cioè algoritmi di alto livello che possono essere applicati a qualsiasi dominio purché esista una procedura di verifica della consistenza per quel dominio (vedi Dravnieks (1991)). La soluzione di base per usi generali per il calcolo di un singolo MUC da un insieme di vincoli logici, consiste nella rimozione iterativa di vincoli da un insiemeini-

ziale. Ad ogni passo, l'insieme di vincoli rappresenta una sovra-approssimazione del MUC. Questa soluzione è indicata come *approccio basato sulla cancellazione* (vedi Dravnieks (1991); Chinneck and Dravnieks (1991); Bakker et al. (1993); Desrosiers et al. (2009)). Dato un insieme R di n vincoli, l'approccio basato sulla cancellazione chiama il controllo di consistenza esattamente n volte. Quando si esamina l'i-esimo vincolo, se $R \setminus \{r_i\}$ rimane inconsistente, allora c'è un MUC che non include r_i e r_i può essere rimosso; altrimenti r_i deve essere parte del MUC. Questo approccio è garantito produrre un insieme $M \subseteq R$ tale che, se un singolo requisito viene eliminato da M, allora M diventa coerente. Tuttavia, l'approccio non garantisce che un altro MUC $M' \subseteq R$ tale che $|M'| \leq |M|$ potrebbe non esistere. La maggior parte degli algoritmi presentati in letteratura sono specifici del dominio (vedi Liffiton and Malik (2013); Marques-Silva and Lynce (2011); Liffiton and Sakallah (2008); Belov and Marques-Silva (2012)) e, per quanto ne sappiamo, nessun approccio specifico che funzioni per LTL è stato finora proposto. L'estrazione di tutti i MUC ha ricevuto una certa attenzione, anche perché la scoperta dei MUC di dimensioni minime può essere fatto semplicemente elencando tutti i MUC. Trovare tutti i MUC di un insieme di vincoli R in un modo *naive* equivale a verificare la consistenza di tutti gli elementi dell'insieme 2^R, ma questo è chiaramente insostenibile nelle applicazioni del mondo reale. In Liffiton and Malik (2013), l'insieme dei requisiti è implicitamente considerato come segue. Dato un insieme di requisiti R, se $R' \subseteq R$ è inconsistente, ogni $R'' \supset R'$ e $R'' \subset R$ è anche inconsistente. Inoltre se $R' \subseteq R$ è consistente, anche ogni $R'' \subset R'$ è consistente. Questo algoritmo può essere modificato per trovare un singolo MUC fermandolo al primo MUC estratto.

Capitolo 3

Analisi dei requisiti di CPS

3.1 Requisiti come Modelli di Specifica della Proprietà (PSP)

Iniziamo definendo un *sistema di vincoli* $\mathcal{D}$ come una tupla $\mathcal{D} = (D, R_1, \ldots, R_n, \mathcal{I})$, dove D è un insieme non vuoto chiamato *dominio* e ogni R_i è un simbolo di predicato di arità a_i, con $\mathcal{I}(R_i) \subseteq D^{a_i}$ essendo la sua interpretazione. Dato un insieme finito di variabili X e un insieme finito di costanti C tale che $C \cap X = \emptyset$, un *termine* è un membro dell'insieme $T = C \cup X$; un *vincolo* (atomico) in $\mathcal{D}$ su un insieme di termini è della forma $R_i(t_1, \ldots, t_{a_i})$ per alcuni $1 \leq i \leq n$ e $t_j \in T$ per tutti $1 \leq j \leq a_i$ che chiamiamo *vincolo* quando $\mathcal{D}$ viene capito dal contesto. Definiamo *Logica Temporale Lineare Modulo Vincoli* – LTL($\mathcal{D}$) in breve – come estensione di LTL con ulteriori vincoli atomici. Dato un insieme di proposizioni booleane $Prop$, un sistema di vincoli $\mathcal{D} = (D, R_1, \ldots, R_n, \mathcal{I})$ e un insieme di termini $T = C \cup X$, una formula LTL($\mathcal{D}$) è definita come:

$$\phi = p \mid R_i(t_1, \ldots, t_{a_i}) \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathcal{X}\,\phi_1 \mid \phi_1\,\mathcal{U}\,\phi_2 \mid (\phi)$$

dove $p \in Prop$, ϕ, ϕ_1, ϕ_2 sono formule LTL($\mathcal{D}$) e $R_i(\cdot)$ con $1 \leq i \leq n$ è un vincolo atomico in $\mathcal{D}$. Ulteriori operatori booleani e temporali sono definiti come in LTL con lo stesso significato. Si noti che l'insieme delle formule LTL($\mathcal{D}$) è un sottoinsieme (rigoroso) di quelli nella logica temporanea lineare con vincoli – CLTL($\mathcal{D}$) in breve – come definito, ad esempio, in Demri and D'Souza (2007). Le formule LTL($\mathcal{D}$) sono interpretate su calcolii della forma $\pi : \mathbb{N} \to 2^{Prop}$ più *valutazioni* aggiuntive della forma $\nu : T \times \mathbb{N} \to D$ tale che, per tutti $i \in \mathbb{N}$, $\nu(c, i) = \nu(c) \in D$ per tutti $c \in C$, mentre $\nu(x, i) \in D$ per tutti $x \in X$. A parole, la funzione ν associa alle costanti $c \in C$ un valore $\nu(c)$ che non cambia nel tempo e alle variabili $x \in X$ un valore

$\nu(x, i)$ che probabilmente cambia ad ogni istante di tempo $i \in \mathbb{N}$. La semantica LTL viene estesa alla LTL($\mathcal{D}$) gestendo i vincoli:

$$\pi, \nu, j \models_{\mathcal{D}} R_i(t_1, \ldots, t_{a_i}) \text{ iff } (\nu(t_1, j), \ldots, \nu(t_{a_i}, j)) \in \mathcal{I}(R_i)$$

Diciamo che π e ν *soddisfano* una formula ϕ, indicata con $\pi, \nu \models_{\mathcal{D}} \phi$, se e solo se $\pi, \nu, 0 \models \phi$. Una formula ϕ è soddisfacibile finché esiste una computazione π e una valutazione ν $\pi, \nu \models_{\mathcal{D}} \phi$. Limitiamo ulteriormente la nostra attenzione al sistema di vincoli $D_C = (\mathbb{R}, <, =, \mathcal{I})$, con vincoli atomici della forma $x < c$ e $x = c$, dove c è una costante corrispondente ad un numero reale – qui di seguito si abusa di notazione e viene scritto $c \in \mathbb{R}$ invece di $\nu(c) \in \mathbb{R}$ – e l'interpretazione $\mathcal{I}$ dei predicati "$<$" e "$=$" è quella solita. Mentre CLTL($\mathcal{D}$) è in generale indecidibile (vedi Demri and D'Souza (2007); Comon and Cortier (2000)), LTL($\mathcal{D}_C$) è decidibile poiché, come mostrato in questo lavoro, può essere ridotto alla soddisfacibilità LTL.

Introduciamo il concetto di *modello di specifica delle proprietà dei vincoli*, indicato con PSP($\mathcal{D}$), per trattare le specifiche contenenti variabili booleane e atomi da un sistema di vincoli $\mathcal{D}$. In particolare, una PSP($\mathcal{D}_C$) presenta solo variabili booleane e vincoli atomici della forma $x < c$ o $x = c$ ($c \in \mathbb{R}$). Per esempio, il requisito:

The angle of joint1 shall never be greater than 170 degrees

può essere riscritto come un PSP($\mathcal{D}_C$):

Globally, it is always the case that $\theta_1 < 170$

dove $\theta_1 \in \mathbb{R}$ è la variabile associata all'angolo di *joint1* e 170 è la soglia limite. Mentre i PSP di base consentono nella loro descrizione solo stati/eventi booleani, i PSP($\mathcal{D}_C$) consentono anche vincoli atomici numerici. È semplice estendere la traduzione di Dwyer et al. (1999) dai PSP di base a LTL per codificare ogni PSP($\mathcal{D}_C$) in una formula in LTL($\mathcal{D}_C$). Si consideri, ad esempio, l'insieme di requisiti:

R_1 Globally, it is always the case that $\mathbf{v} \leq \mathbf{5.0}$ holds.

R_2 After $\mathbf{a}$, $\mathbf{v} \leq \mathbf{8.5}$ eventually holds.

R_3 After $\mathbf{a}$, it is always the case that if $\mathbf{v} \geq \mathbf{3.2}$ holds, then $\mathbf{z}$ eventually holds.

dove $\mathbf{a}$ e $\mathbf{z}$ sono stati/eventi booleani, mentre $\mathbf{v}$ è un segnale numerico. Questi PSP($\mathcal{D}_C$) possono essere riscritti come nella seguente formula LTL($\mathcal{D}_C$):

$$\begin{aligned} &\Box(v < 5.0 \vee v = 5.0) && \wedge \\ &\Box(a \rightarrow \Diamond(v < 8.5) \vee (v = 8.5)) && \wedge \\ &\Box(a \rightarrow \Box(\neg(v < 3.2) \rightarrow \Diamond z)) \end{aligned} \tag{3.1}$$

Pertanto, ragionare sulla consistenza dell'insieme dei requisiti scritti usando PSP($\mathcal{D}_C$) è sufficiente fornire un algoritmo per decidere la soddisfacibilità di formule LTL($\mathcal{D}_C$).

A tal fine, consideriamo una formula ϕ del tipo LTL($\mathcal{D}_C$), e sia $X(\phi)$ l'insieme delle variabili e sia $C(\phi)$ l'insieme delle costanti che si verificano in ϕ. Definiamo l'*insieme delle soglie* $S_x(\phi) \subseteq C(\phi)$ come l'insieme di valori costanti rispetto al quale viene confrontata una variabile $x \in X(\phi)$; più precisamente, per ogni variabile $x \in X(\phi)$ costruiamo un insieme $S_x(\phi) = \{c_1, .., c_n\}$ tale che, per tutti i $c_k \in \mathbb{R}$ con $1 \leq k \leq n$, ϕ, ϕ contiene un vincolo della forma $x < c_k$ or $x = c_k$. Per comodità, consideriamo sempre ogni insieme di soglie $S_x(\phi)$ ordinata in ordine ascendente, cioè $c_k < c_{k+1}$ per tutti $1 \leq k < n$. Ad esempio, nell'esempio (3.1), abbiamo $X = \{v\}$ e il corrispondente insieme di soglia è $S_v = \{3.2, 5.0, 8.5\}$. Data una formula ϕ del tipo LTL($\mathcal{D}_C$), e alcune variabili $x \in X(\phi)$, sia $S_x(\phi) = \{c_1, \ldots, c_n\}$ l'insieme delle soglie per le quali definiamo i corrispondenti insiemi di *proposizioni di disuguaglianza* $Q_x(\phi) = \{q_1, \ldots, q_n\}$ e di *proposizioni di uguaglianza* $E_x(\phi) = \{e_1, \ldots, e_n\}$. Informalmente, le proposizioni di disuguaglianza dovrebbero essere vere esattamente quando una variabile $x \in X(\phi)$ è sotto o tra qualche valore nell'insieme delle soglie $S_x(\phi)$, mentre le proposizioni di uguaglianza dovrebbero essere vere esattamente quando x è uguale a qualche valore in $S_x(\phi)$. A causa di ciò, nella nostra codifica dobbiamo assicurarci che per ogni computazione π e istante temporale $i \in \mathbb{N}$ esattamente uno dei seguenti casi è vero ($1 \leq j \leq n$):

- $q_j \in \pi(i)$ for some j, $q_l \notin \pi(i)$ for all $l \neq j$ and $e_j \notin \pi(i)$ for all j;
- $e_j \in \pi(i)$ for some j, $e_l \notin \pi(i)$ for all $l \neq j$ and $q_j \notin \pi(i)$ for all j;
- $q_j \notin \pi(i)$ and $e_j \notin \pi(i)$ for all j.

Il primo caso sopra corrisponde a un valore di x che si trova tra alcuni valori delle soglie in $S_x(\phi)$ o prima del suo valore più piccolo; il secondo caso si verifica quando un valore di soglia è uguale a x e il terzo caso si ha quando x supera il massimo valore di soglia in $S_x(\phi)$.

Date le definizioni sopra, una formula ϕ di tipo LTL($\mathcal{D}_C$) sull'insieme di proposizioni booleane $Prop$ e l'insieme di termini $T = C \cup X$, possono essere convertiti in una formula LTL ϕ' sull'insieme di proposizioni booleane $Prop \cup \bigcup_{xinX}(Q_x(\phi) \cup E_x(\phi))$. Otteniamo questo considerando, per ogni variabile $x \in X$ e per l'insieme di soglie associate $S_x(\phi)$, le corrispondenti proposizioni $Q_x(\phi) = \{q_1, \ldots q_n\}$ e $E_x = \{e_1, \ldots, e_n\}$; quindi, per ogni $c_k \in S_x(\phi)$, eseguiamo le seguenti sostituzioni:

$$x < c_k \rightsquigarrow \bigvee_{j=1}^{k} q_j \vee \bigvee_{j=1}^{k-1} e_j \qquad \text{and} \qquad x = c_k \rightsquigarrow e_k. \tag{3.2}$$

Sostituendo i vincoli numerici atomici non è sufficiente per garantire l'equisoddisfacibilità di ϕ' rispetto a ϕ. In particolare, per ogni $x \in X(\phi)$, dobbiamo codificare l'osservazione informale fatta in precedenza sulle valutazioni booleane "mutualmente esclusive" per proposizioni in $Q_x(\phi)$ ed in $E_x(\phi)$ come vincoli corrispondenti:

$$\phi_M = \bigwedge_{x \in X(\phi)} \left(\bigwedge_{a,b \in M_x(\phi), a \neq b} \Box\neg(a \wedge b) \right) \tag{3.3}$$

dove $M_x(\phi) = Q_x(\phi) \cup E_x(\phi)$. Per esempio, dato l'esempio (3.1), $Q_v = \{q_1, q_2, q_3\}$ e $E_v = \{e_1, e_2, e_3\}$ e i vincoli di mutua esclusione sono scritti come:

$$\begin{aligned} \phi_M = & \Box\neg(q_1 \wedge q_2) \wedge \Box\neg(q_1 \wedge q_3) \wedge \Box\neg(q_1 \wedge e_1) \wedge \Box\neg(q_1 \wedge e_2) \wedge \\ & \Box\neg(q_1 \wedge e_3) \wedge \Box\neg(q_2 \wedge q_3) \wedge \Box\neg(q_2 \wedge e_1) \wedge \Box\neg(q_2 \wedge e_2) \wedge \\ & \Box\neg(q_2 \wedge e_3) \wedge \Box\neg(q_3 \wedge e_1) \wedge \Box\neg(q_3 \wedge e_2) \wedge \Box\neg(q_3 \wedge e_3) \wedge \\ & \Box\neg(e_1 \wedge e_2) \wedge \Box\neg(e_1 \wedge e_3) \wedge \Box\neg(e_2 \wedge e_3). \end{aligned} \tag{3.4}$$

Pertanto, la formula LTL da testare per valutare la consistenza dei requisiti è

$$\begin{aligned} \phi_M \wedge (\quad & \Box(q_1 \vee q_2 \vee e_1 \vee e_2) \wedge \\ & \Box(a \rightarrow \Diamond(\textstyle\bigvee_{i=1}^{3} q_i \vee e_i)) \wedge \\ & \Box(a \rightarrow \Box(\neg q_1 \rightarrow \Diamond z))). \end{aligned} \tag{3.5}$$

Possiamo ora affermare quanto segue:

Teorema 1 *Sia ϕ una formula LTL($\mathcal{D}_C$) sull'insieme di proposizione Prop e termini $T = X(\phi) \cup C(\phi)$; per ogni $x \in X(\phi)$, sia $S_x(\phi)$, $Q_x(\phi)$ e $E_x(\phi)$ il corrispondente insieme di soglie, le proposizioni di disuguaglianza e le proposizioni di uguaglianza rispettivamente; Sia ϕ' la formula LTL sull'insieme di proposizione $Prop \cup \bigcup_{x \in X(\phi)} Q_x(\phi) \cup E_x(\phi)$ ottenuto da ϕ applicando le sostituzioni (3.2) per ogni $x \in X(\phi)$ e $c_k \in S_x(\phi)$, e sia ϕ_M la formula LTL ottenuta come in (3.3); quindi, la formula ϕ del tipo LTL($\mathcal{D}_C$) è soddisfacibile se e solo se la formula LTL $\phi_M \wedge \phi'$ è soddisfacibile.*

Anche la traduzione proposta da LTL($\mathcal{D}_C$) ad una formula LTL è abbastanza compatta, cioè il numero di simboli nella codifica LTL cresce al massimo quadraticamente con il numero di simboli nella formula originale. Cerchiamo di definire la dimensione di una formula ϕ, denotata come $|\phi|$, nel solito modo, cioè contando il numero di simboli in essa. Possiamo affermare quanto segue:

r_i	PSP
r_1	Globally, it is always the case that A holds.
r_2	Globally, it is never the case that A holds.
r_3	Globally, it is always the case that B holds.
r_4	Globally, it is always the case that if B holds, then C holds as well.
r_5	Globally, it is never the case that C holds.
r_6	Globally, it is always the case that A and B holds.
r_7	After B, D eventually holds.

Tabella 3.1: Insieme R di PSP inconsistenti.

Teorema 2 *Sia ϕ una formula LTL($\mathcal{D}_C$) sull'insieme di proposizione Prop e termini $T = X(\phi) \cup C(\phi)$; per ogni $x \in X(\phi)$, sia $S_x(\phi)$, $Q_x(\phi)$ e $E_x(\phi)$ il corrispondente insieme di soglie, le proposizioni di disuguaglianza e le proposizioni di uguaglianza rispettivamente; sia ϕ' la formula LTL sull'insieme di proposizione on the set of proposition $Prop \cup \bigcup_{x \in X(\phi)} Q_x(\phi) \cup E_x(\phi)$ ottenuta da ϕ applicando le sostituzioni (3.2) per ogni $x \in X(\phi)$ e $c_k \in S_x(\phi)$, e sia ϕ_M la formula LTL ottenuta come in (3.3); la dimensione di $\phi' \wedge \phi_M$ è al più quadratica nella dimensione di ϕ, cioè $O(|\phi' \wedge \phi_M|) = O(|\phi|^2)$.*

3.2 Verifica dell'inconsistenza

Dato un insieme $R = \{r_1, \ldots, r_n\}$ di requisiti incoerenti scritti come PSP($\mathcal{D}_C$), l'obiettivo degli algoritmi proposti in questa sezione è quello di calcolare un *Nucleo Minimo Insoddisfacibile* (*MUC*), cioè un sottoinsieme $I \subseteq R$ tale che rimuovendo qualsiasi elemento r_i da I rende l'insieme di nuovo consistente. La tabella 3.1 mostra una specifica inconsistente come un insieme $R = \{r_1, \ldots r_7\}$ di sette requisiti. Guardando la tabella, possiamo vedere che ci sono 4 MUC differenti in R, cioè $\{r_1, r_2\}$, $\{r_2, r_6\}$, $\{r_3, r_4, r_5\}$, $\{r_4, r_5, r_6\}$. Nel resto della sezione presentiamo due algoritmi dedicati all'estrazione di MUC per PSP.

3.2.1 Estrazione di un MUC basata su cancellazione lineare

Il primo algoritmo che presentiamo considera una strategia basata sull'eliminazione e il suo pseudo-codice è rappresentato nell'Algoritmo 3.1. La procedura funziona come segue. Se l'insieme $R' \leftarrow R \setminus \{r\}$ con $r \in R$ è inconsistente, allora r non è nel MUC. D'altra parte, se R' è consistente, allora r fa parte di un MUC e non può essere rimosso. Tale operazione viene ripetuta in modo iterativo e l'algoritmo termina quando tutti i requisiti sono stati verificati per l'inclusione nel MUC.

Algoritmo 3.1 Algoritmo di estrazione di un MUC basato su cancellazione lineare

```
1: function findInconsistency(R)
2:     R' ← R
3:     for r_i ∈ R do
4:         R' ← R' \ {r_i}
5:         if isConsistent(R') then
6:             R' ← R' ∪ {r_i}
7:         end if
8:     end for
9:     return R'
10: end function
```

È facile vedere che, con $|R| = n$, il ciclo *for* itera n volte e che ad ogni iterazione la funzione isConsistent viene chiamata una volta. L'input della funzione è R' e le sue dimensioni sono date da $|R'|$. Il numero di elementi in R' è ridotto di uno ad ogni iterazione, ma r_i potrebbe essere nuovamente aggiunto in R', a seconda del risultato della funzione isConsistent. Il caso peggiore si ottiene quando tutti i requisiti fanno parte del MUC, vale a dire, ogni requisito r_i viene prima rimosso e quindi reinserito nuovamente. In questo caso il model checker viene chiamato ogni volta con $n - 1$ requisiti. La complessità complessiva è quindi $O(n \cdot C(n))$, dove n è il numero di elementi iniziali in R e $C(n)$ è la complessità per la verifica della consistenza di n requisiti. L'algoritmo è quindi lineare nel numero di chiamate al model checker.

Esempio 1 *Considerando l'insieme R nella Tabella 3.1, l'Algoritmo 3.1 funziona seguendo i seguenti passaggi.*

Step	r_i	R'	isConsistent(R')
1:	r_1	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$	FALSE
2:	r_2	$\{r_3, r_4, r_5, r_6, r_7\}$	FALSE
3:	r_3	$\{r_4, r_5, r_6, r_7\}$	FALSE
4:	r_4	$\{r_5, r_6, r_7\}$	TRUE
5:	r_5	$\{r_4, r_6, r_7\}$	TRUE
6:	r_6	$\{r_4, r_5, r_7\}$	TRUE
7:	r_7	$\{r_4, r_5, r_6\}$	FALSE

Il risultato finale è $R' = \{r_4, r_5, r_6\}$. Vale la pena notare che questo risultato dipende sull'ordine di estrazione dei requisiti. È facile vedere che elaborando i requisiti in ordine inverso produrrà invece $R' = \{r_1, r_2\}$.

3.2.2 Estrazione di un MUC in maniera dicotomica

L'Algoritmo 3.2 è basato sulla stessa struttura generale dell'Algoritmo 3.1, ma sfrutta anche il fatto che la dimensione del MUC è spesso molto più piccola di $|R|$. Pertanto, è possibile sfruttare una strategia di "dividi e conquista" per ridurre lo spazio di ricerca. Considerando l'Algoritmo 3.2, R viene diviso in due metà R_1 e R_2, tale che $R_1 \cup R_2 = R$ e $R_1 \cap R_2 = \emptyset$. Se una delle due metà (più I) è inconsistente, allora non c'è bisogno di esplorare l'altro e possiamo procedere in modo ricorsivo. Altrimenti significa che il MUC è stato diviso in due metà e non sono necessarie ulteriori ricerche. Questo viene fatto per mezzo di due chiamate ricorsive (righe 21–22); La prima esegue la ricerca su R_2 considerando l'intero insieme R_1 come inconsistente, mentre la seconda continua la ricerca su R_1 , rimuovendo da I i requisiti che devono ancora essere controllati. L'algoritmo termina quando R ha 1 o 0 elementi.

Algoritmo 3.2 Algoritmo di estrazione di un MUC in maniera dicotomica

```
1:  function findInconsistency(R)
2:      return findInconsistency(R, ∅)
3:  end function

4:  function findInconsistency(R, I)
5:      if |R| ≤ 1 then
6:          if isConsistent(I) then
7:              return I ∪ R
8:          else
9:              return I
10:         end if
11:     end if
12:     (R1, R2) ← split(R)
13:     if |R1| > 1 and |R2| > 1 then
14:         if ¬ isConsistent(R1 ∪ I) then
15:             return findInconsistency(R1, I)
16:         end if
17:         if ¬ isConsistent(R2 ∪ I) then
18:             return findInconsistency(R2, I)
19:         end if
20:     end if
21:     I ← findInconsistency(R2, I ∪ R1)
22:     I ← findInconsistency(R1, I \ R1)
23:     return I
24: end function
```

Per quanto riguarda la complessità dell'algoritmo, il caso migliore si verifica quando il MUC è sempre nella prima metà di R. In tal caso, metà dei requisiti viene scartata ad ogni iterazione, ed è facile vedere che la complessità è $\Omega(\log |R|)$. Il peggior caso si verifica quando l'insieme di requisiti inconsistenti coincide con R. Prendendo in considerazione la Tabella 3.1, sia R composto da $\{r_1, r_2, r_3, r_4\}$ e lasciamo che il MUC sia R. Al primo passo, l'algoritmo controlla $R'_1 = \{r_1, r_2\}$ e $R'_2 = \{r_3, r_4\}$ ma entrambi gli insiemi sono consistenti. Pertanto, FINDINCONSISTENCY viene chiamata in modo ricorsivo con $R = \{r_3, r_4\}$ e $I = \{r_1, r_2\}$. A questo punto abbiamo $R''_1 = \{r_3\}$ e $R''_2 = \{r_4\}$. L'algoritmo controlla la consistenza di $\{r_1, r_2, r_3\}$ e $\{r_1, r_2, r_4\}$ e ritorna alla precedente chiamata ricorsiva. Questa volta FINDINCONSISTENCY viene richiamata di nuovo, ma con $R = \{r_1, r_2\}$ e $I = \{r_3, r_4\}$ e viene applicato lo stesso processo. In generale, se $|R| = n$ e $C(n)$ è la complessità per il controllo della consistenza di n requisiti, allora la complessità nel caso peggiore di questo algoritmo è $O(n \cdot C(n))$ – la stesso del precedente. Tuttavia quando $|I| \ll |R|$ è notevolmente più veloce della versione lineare.

Esempio 2 *Considerando nuovamente l'insieme R riportato nella Tabella 3.1, di seguito riportiamo passo-passo come funziona l'Algoritmo 3.2. Per mancanza di spazio nella tabella sostituiamo* ISCONSISTENT *con isCons*

Step	R	R_1	R_2	I	$isCons(R_1 \cup I)$	$isCons(R_2 \cup I)$
1:	$\{r_1, \ldots, r_7\}$	$\{r_1, r_2, r_3\}$	$\{r_4, r_5, r_6, r_7\}$	$\{\}$	$False$	–
2:	$\{r_1, r_2, r_3\}$	$\{r_1\}$	$\{r_2, r_3\}$	$\{\}$	$True$	$True$
3:	$\{r_2, r_3\}$	$\{r_2\}$	$\{r_3\}$	$\{r_1\}$	–	–
4:	$\{r_2\}$	–	–	$\{r_1\}$	–	–
5:	$\{r_1\}$	–	–	$\{r_2\}$	–	–

Nel primo passo, l'algoritmo divide l'insieme iniziale R in due sottoinsiemi R_1 e R_2 e controlla la consistenza del primo. Poiché R_1 è inconsistente, l'algoritmo elimina automaticamente R_2 e continua con il passo 2. Anche in questo caso il nuovo insieme $R = \{r_1, r_2, r_3\}$ è diviso in due, ma questa volta entrambi sono consistenti e così vengono eseguite le due chiamate ricorsive nelle righe 21-22: la prima è risolta nei passi 3 e 4, mentre la seconda nel passo 5. Negli ultimi due passi, il caso base è stato raggiunto (righe 5-11) e poiché la chiamata a ISCONSISTENT(I) *restituisce true in entrambi i casi, r_1 e r_2 vengono aggiunti a I. Pertanto, $I = \{r_1, r_2\}$ viene restituito come risposta finale. In questo caso* ISCONSISTENT *viene chiamato 6 volte anziché 7 come nell'esempio precedente.*

Capitolo 4

Sommario

In questo documento, sono state presentate le tecniche di verifica dei sistemi, e in particolare la verifica automatica del modello (Model Checking). Inoltre sono stati presentati i Modelli di Specifica della Proprietà (PSP) e l'analisi dei requisiti di un CPS descritti come modelli PSP. Inoltre abbiamo esteso i PSP di base sul sistema di vincoli $\mathcal{D}_C$,e abbiamo fornito una codifica da qualsiasi PSP($\mathcal{D}_C$) in una formula LTL corrispondente. Questa codifica consente di affrontare la soddisfacibilit'‘a delle specifiche di interesse pratico, e di verificarle utilizzando strumenti di ragionamento automatico allo stato dell'arte disponibili per LTL.

Bibliografia

R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI*, volume 93, pages 276–281, 1993.

J. Barnat, P. Bauch, N. Beneš, L. Brim, J. Beran, and T. Kratochvíla. Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing*, 28(1): 45–63, 2016.

A. Belov and J. Marques-Silva. Accelerating mus extraction with recursive model rotation. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 37–40. IEEE, 2011.

A. Belov and J. Marques-Silva. Muser2: An efficient mus extractor. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:123–128, 2012.

J. Bendík. Consistency checking in requirements analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 408–411. ACM, 2017.

J. W. Chinneck and E. W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *14th International Conference on Computer Aided Verification (CAV 2002)*, pages 359–364, 2002.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

H. Comon and V. Cortier. Flatness is not a weakness. In *International Workshop on Computer Science Logic*, pages 262–276, 2000.

S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, 2007.

C. Desrosiers, P. Galinier, A. Hertz, and S. Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *Journal of combinatorial optimization*, 18(2):124–150, 2009.

L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):131–165, 1994.

E. W. Dravnieks. Identifying minimal sets of inconsistent constraints in linear programs: Deletion, squeeze and sensitivity filtering. 1991.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International conference on Software engineering*, pages 411–420, 1999.

U. Hustadt and B. Konev. TRP++ 2.0: A temporal resolution prover. In *19th International Conference on Automated Deduction*, pages 274–278, 2003.

U. Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, 2001.

Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *International Conference on Computer Aided Verification*, pages 97–109. Springer, 1993.

S. Konrad and B. H. Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381, 2005.

J. Li, G. Pu, L. Zhang, Y. Yao, M. Y. Vardi, et al. Polsat: A portfolio LTL satisfiability solver. *arXiv preprint arXiv:1311.1602*, 2013a.

J. Li, L. Zhang, G. Pu, M. Y. Vardi, and J. He. LTL satisfiability checking revisited. In *20th International Symposium on Temporal Representation and Reasoning*, pages 91–98, 2013b.

J. Li, Y. Yao, G. Pu, L. Zhang, and J. He. Aalta: an LTL satisfiability checker over infinite/finite traces. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 731–734, 2014.

J. Li, S. Zhu, G. Pu, and M. Y. Vardi. Sat-based explicit ltl reasoning. In *11th Haifa Verification Conference*, pages 209–224, 2015.

M. H. Liffiton and A. Malik. Enumerating infeasibility: Finding multiple muses quickly. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 160–175. Springer, 2013.

M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

J. Marques-Silva and I. Lynce. On improving mus extraction algorithms. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 159–173. Springer, 2011.

M. Masin, F. Palumbo, H. Myrhaug, J. de Oliveira Filho, M. Pastena, M. Pelcat, L. Raffo, F. Regazzoni, A. Sanchez, A. Toffetti, et al. Cross-layer design of reconfigurable cyber-physical systems. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 740–745. IEEE, 2017.

A. Nadel. Boosting minimal unsatisfiable core extraction. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 221–229. FMCAD Inc, 2010.

A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

A. Pnueli and Z. Manna. The temporal logic of reactive and concurrent systems. *Springer*, 16:12, 1992.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.

A. Post and J. Hoenicke. Formalization and analysis of real-time requirements: A feasibility study at BOSCH. *Verified Software: Theories, Tools, Experiments*, pages 225–240, 2012.

K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Spin*, volume 4595, pages 149–167. Springer, 2007.

K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):123–137, 2010.

K. Y. Rozier and M. Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *International Symposium on Formal Methods*, pages 417–431. Springer, 2011.

S. Schwendimann. A new one-pass tableau calculus for pltl. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 277–291. Springer, 1998.

P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, pages 119–136, 1985.