VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

DETEKCE ANOMÁLIÍ BĚHU RTOS APLIKACE

DIZERTAČNÍ PRÁCE
DOCTORAL THESIS

AUTOR PRÁCE Ing. JAKUB ARM
AUTHOR

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

DETEKCE ANOMÁLIÍ BĚHU RTOS APLIKACE

DETECTING RTOS RUNTIME ANOMALIES

DIZERTAČNÍ PRÁCE
DOCTORAL THESIS

AUTOR PRÁCE — Ing. Jakub Arm
AUTHOR

ŠKOLITEL — doc. Ing. Zdeněk Bradáč, Ph.D.
SUPERVISOR

BRNO 2020

ABSTRAKT

S vyššími požadavky na výpočetní výkon a bezpečnost (resp. funkční bezpečnost) zařízení v průmyslové doméně jsou vestavné systémy spolu s operačními systémy reálného času stále předmětem výzkumu. Tato práce se zabývá kontrolním subsystémem běhu softwarového vybavení založeným na modelu aplikace, který zlepšuje diagnostické pokrytí chyb zejména anomálií vykonávání RTOS. Po specifikaci architektury tohoto subsystému následuje formální definice modelu a jeho implementace do hardware, resp. FPGA. Práce popisuje i další možné směry výzkumu a také přináší nové pohledy na rozebíranou problematiku, např. kombinaci s návrhovými vzory. Nedílnou součástí je i ověření funkčnosti navrhnutého modulu pomocí simulace na testovacích scénářích, které vychází ze změřeného záznamu událostí reálné aplikace. Z výsledků vyplývá, že vyvinutý modul dosahuje řádově nižšího času detekce než standardní watchdog.

KLÍČOVÁ SLOVA

Detekce anomálií, Funkční bezpečnost, Petriho síť, operační systém reálného času, kontrola běhu programu, návrhový vzor, FPGA

ABSTRACT

Due to higher requirements of computational power and safety, or functional safety of equipments intended for the use in the industrial domain, embedded systems containing a real-time operating system are still the active area of research. This thesis addresses the hardware-assisted control module that is based on the runtime model-based verification of a target application. This subsystem is intended to increase the diagnostic coverage, particularly, the detection of the execution errors. After the specification of the architecture, the formal model is defined and implemented into hardware using FPGA technology. This thesis also discuss some other aspects and embodies new approaches in the area of embedded flow control, e.g. the integration of the design patterns. Using the simulation, the created module was tested using the created scenarios, which follow the real program execution record. The results suggest that the error detection time is lower than using standard techniques, such a watchdog.

KEYWORDS

Anomaly detection, Functional safety, Petri net, RTOS, program flow control, design pattern, FPGA

ARM, Jakub *Detekce anomálií běhu RTOS aplikace*: dizertační práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky, 2020. 128 s. Vedoucí práce byl doc. Ing. Zdeněk Bradáč, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou doktorskou práci na téma „Detekce anomálií běhu RTOS aplikace“ jsem vypracoval samostatně pod vedením vedoucího doktorské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené doktorské práce dále prohlašuji, že v souvislosti s vytvořením této doktorské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

(podpis autora)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu disertační práce panu doc. Ing. Zdeňku Bradáčovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

(podpis autora)

OBSAH

Úvod 11

1 Cíle práce 13

1.1 Předmět práce 13

1.2 Očekávané využití dosažených výsledků 14

1.3 Stanovení hypotéz 15

1.4 Členění dokumentu 15

2 Teoretický úvod 16

2.1 Operační systémy reálného času 16

2.1.1 Definice 16

2.1.2 Dělení 16

2.1.3 Popis 17

2.1.4 Chyby v RTOS 18

2.1.5 Více-procesorové systémy 19

2.1.6 Vlastnosti 19

2.2 Funkční bezpečnost 20

2.2.1 Definice pojmů 20

2.2.2 Zdroje chyb 21

2.2.3 Metody boje proti chybám 22

2.2.4 Standard IEC 61508 28

3 Literární rozbor 29

3.1 Literární prameny 29

3.2 Inovátorská řešení 30

3.2.1 Yogitech's faultRobust-CPU 30

3.2.2 Mikrokontroléry Hercules 30

3.2.3 Hardwarový plánovač pro detekci chyb 31

3.2.4 Hardwarový multi-watchdog 31

3.2.5 Runtime měření času vykonání vláken 32

3.2.6 Diagnostický softwarový systém 32

3.2.7 Monitor řešitelnosti plánovače 32

3.2.8 Řízení dle scénáře 32

3.2.9 Softwarová řešení 33

3.2.10 Implementace specifikace v hardware 33

3.2.11 Hardware kontrolní subsystém 33

3.2.12 RTOS-G 33

4 Formální metody **35**
4.1 Temporální logika 36
4.2 UML 36
4.3 AADL 37
4.4 Matlab Simulink 38
4.5 Přechodové systémy 38
4.6 VHDL 39
4.7 Petriho sítě 39
4.8 Zhodnocení 41

5 Hardwarová implementace algoritmů **42**
5.1 Rozhraní testovacího systému 42
5.1.1 ARM Cortex CoreSight a JTAG 42
5.1.2 Soft-core 43
5.1.3 Koprocesor 44
5.1.4 Standardní komunikační sběrnice mikroprocesoru 44
5.1.5 Software tracing 44
5.2 Hardwarová implementace algoritmů 45
5.2.1 Možnosti prostředí FPGA 45
5.3 Výběr soft-core 46
5.3.1 Plasma 46
5.3.2 Leon 47
5.3.3 MicroBlaze 47
5.3.4 PicoBlaze 47
5.3.5 Nios II 48
5.4 Zhodnocení 48

6 Návrhové vzory **49**
6.1 Popis 49
6.2 Použití 50
6.3 Rozdělení 51
6.4 Problémy 52
6.5 Metody notace a formalizace 53
6.5.1 LayOM 54
6.5.2 Atributy 54
6.5.3 LePUS 54
6.5.4 Jazyk typu Z 54
6.5.5 UML 54
6.5.6 Nástroje pro práci s návrhovými vzory 55

6.5.7 Model vs. návrhový vzor vs. vzor architektury vs. idiom . . . 55
6.6 Návrhové vzory 56
6.6.1 Vzory architektury systému 57
6.6.2 Vzory v paralelně vykonávaných systémech 57
6.6.3 Vzory obsluhy paměťového prostoru 57
6.6.4 Vzory distribuce dat 57
6.6.5 Vzory paralelního běhu úloh 57
6.7 Zhodnocení 58

7 Měření reálných scénářů 59
7.1 Prostředí a nástroje 59
7.2 Měřicí systém 60
7.3 Testovací scénáře 61
7.3.1 Deadlock 61
7.3.2 Livelock 62
7.3.3 Starvation 62
7.3.4 Race condition 63
7.4 Zhodnocení 64

8 Online kontrolní systém 65
8.1 Cíle online kontrolního systému 65
8.2 Popis online kontrolního systému 65
8.2.1 Modul rozhraní 66
8.2.2 Kontrolní modul 66
8.2.3 Model sledovaného systému 67
8.3 Specifikace modelu 69
8.3.1 Monitorování 70
8.3.2 Kontrola správnosti přechodů 70
8.3.3 Kontrola časových podmínek 71
8.3.4 Volba modelovacího nástroje 71
8.3.5 Formální definice 72
8.3.6 Implementace kontrolního modulu 74
8.3.7 Integrace do vývoje cílové aplikace 75
8.4 Ukázka implementace návrhového vzoru synchronizačního objektu mutex 75
8.5 Definice vzoru architektury 76
8.6 Zhodnocení 77

9 Simulace **78**
9.1 Prostředí a nástroje 78
9.2 Architektura simulace 78
9.3 Testovací scénář 80
9.4 Implementace modelu programu 83
9.5 Výsledky simulace 85
9.5.1 Třída I – normální běh 85
9.5.2 Třída II – časová chyba 86
9.5.3 Třída III – chyba kontinuity 86
9.6 Hodnocení výsledků 86

10 Závěr **88**
10.1 Vyhodnocení hypotéz 90
10.2 Přínosy 91
10.3 Aplikace 92
10.4 Směry pro další výzkum 92

Vlastní publikační činnost - vybrané publikace **93**

Literatura **95**

Seznam symbolů, veličin a zkratek **110**

Seznam příloh **114**

A Přílohy **115**
A.1 Implementace uzlu (místo + hrana) kontrolního modelu ve VHDL . 115
A.2 Integrace procesu vytvoření kontrolního systému do standardního V-modelu 116
A.3 Návrh hardware platformy pro měření událostí pomocí mapované periferie 117
A.4 Implementace scénáře 1 – *deadlock* 118
A.5 Záznam událostí RTOS ze scénáře 1 – *deadlock* 119
A.6 Implementace scénáře 2 – *livelock* 120
A.7 Záznam událostí RTOS ze scénáře 2 – *livelock* 121
A.8 Implementace scénáře 3 – *starvation* 122
A.9 Záznam událostí RTOS ze scénáře 3 – starvation 123
A.10 Implementace scénáře 4 – *race condition* 124
A.11 Záznam událostí RTOS ze scénáře 4 – *race condition* 125
A.12 Záznam událostí simulace třídy I – normální běh 126
A.13 Záznam událostí simulace třídy II – časová chyba 127

A.14 Záznam událostí simulace třídy III – chyba kontinuity 128

SEZNAM OBRÁZKŮ

2.1 Parametry úlohy [22] 18
2.2 Závislost četnosti přechodných chyb na použité technologii [16] 21
2.3 Schéma systému s EUC [85] 23
3.1 Připojení fRCPU na CPU [85] 31
5.1 Schéma technologie *ARM CoreSight* [73] 43
6.1 Šablona pro definici návrhového vzoru podle [5] 50
7.1 Blokový diagram měřicího řetězce 61
8.1 Architektura systému online kontroly systému 66
8.2 Blokové schéma online kontrolního systému 67
8.3 Diagram uzlu 75
8.4 Příklad kontrolní Petriho sítě skládající se z uzlů 75
8.5 Ukázka implementace návrhového vzoru synchronizačního objektu mutex 76
9.1 Blokové schéma testovací architektury 80
9.2 Blokové schéma kontrolního modulu 81
9.3 Definice modulu *PetriNode* v jazyce VHDL 81
9.4 Záznam událostí reálně běžícího scénáře 82
9.5 Výpis časového signálu jako stimulu pro simulaci 83
9.6 Ukázka definice stimulu pro *testbench* v jazyku SystemVerilog 84
9.7 Ukázka definice modelu pomocí *Petri node* v jazyku *Verilog* 84
9.8 Modelování plánovače RTOS jako separátní *kontrolní Petriho síť* . . . 85
A.1 Implementace uzlu (místo + hrana) kontrolního modelu ve VHDL . . 115
A.2 Integrace procesu vytvoření kontrolního systému do standardního V-modelu 116
A.3 Návrh hardware platformy pro měření událostí pomocí mapované periferie 117
A.4 Implementace scénáře 1 – *deadlock* 118
A.5 Záznam událostí RTOS ze scénáře 1 – *deadlock* 119
A.6 Implementace scénáře 2 – *livelock* 120
A.7 Záznam událostí RTOS ze scénáře 2 – *livelock* 121
A.8 Implementace scénáře 3 – *starvation* 122
A.9 Záznam událostí RTOS ze scénáře 3 – *starvation* 123
A.10 Implementace scénáře 4 – *race condition* 124
A.11 Záznam událostí RTOS ze scénáře 4 – *race condition* 125
A.12 Průběh vybraných signálů simulace třídy I – **normální běh** 126
A.13 Průběh vybraných signálů simulace třídy II – **časová chyba** 127
A.14 Průběh vybraných signálů simulace třídy III – **chyba kontinuity** . . 128

ÚVOD

Vestavné (angl. *embedded*) systémy jsou nedílnou součástí života, protože život usnadňují, zpříjemňují a svým způsobem i umožňují. Mnohdy si ani neuvědomujeme jejich přítomnost a přitom vyžadujeme jejich bezporuchový, či dokonce bezpečný chod. Výkon a komplexnost těchto zpravidla jednoúčelových systémů jsou navrhovány dle požadavků na jejich funkci. Ze všech možných oblastí použití se letectví, automobilismus, medicína, průmysl a vojenství oddělují kvůli zvýšeným požadavkům, které jsou na tyto systémy kladeny.

Důležitou společnou vlastností těchto systémů je bezpečnost, a to jak kybernetická bezpečnost, tak funkční bezpečnost. Funkční bezpečnost zajišťuje bezproblémovou integraci vestavných systémů a je nedílnou součástí celkové bezpečnosti takového systému. Zvýšené nároky na úroveň funkční bezpečnosti s sebou nesou vyšší požadavky na spolehlivost systému, propracovanější analýzu funkční bezpečnosti a její posouzení. Tyto systémy jsou potom schopné v určité míře fungovat správně i za přítomnosti chyby, nebo dokonce poruchy a nazývají se systémy odolné proti poruchám (angl. *fault-tolerant systems*). Jsou zpravidla nasazovány v zařízeních plnících bezpečnostní funkce (angl. *safety functions*) nebo zařízeních zajišťujících bezpečný chod systému (angl. *safety-related*). Požadavky na analýzu, implementaci a uvedení do provozu zařízení, resp. obecně systémů se zvýšenými požadavky na funkční bezpečnost, se zabývá obecná norma IEC 61508.

V průmyslové automatizaci nebo i jiných oblastech, jako je aerospace, automotive nebo medical, je velká poptávka po vysoce bezpečných systémech (angl. *safety-critical systems*). Často se právě kvůli požadavku na vysokou spolehlivost sází na dlouhodobě odzkoušená zařízení a řešení. Jelikož vývoj ve výpočetní technice jde mílovými kroky kupředu, tato zařízení nestačí svým výkonem, a tendencí je nasazovat buď víceprocesorové systémy, které spotřebují větší část výpočetního výkonu pro synchronizaci, nebo systémy s vyšším výpočetním výkonem, které jsou ale zároveň zranitelnější ve ztížených provozních podmínkách. Tyto zpravidla dlouhodobě neodzkoušené systémy je tedy nutné opatřit subsystémy nebo použít důmyslnější techniky pro zajištění požadované funkční bezpečnosti.

U systémů, kde je správnost výpočtů a navíc i čas výpočtu kritický, se nasazují tzv. operační systémy reálného času (angl. *real-time operating system*). Tyto systémy musí zajišťovat běh aplikací podle předem daných pravidel (angl. *determinismus*). Jelikož se jedná o komplexní konkurentní operační systém, mohou při jeho provozu vzniknout předvídatelné i nepředvídatelné defekty (anomálie) vedoucí k chybám, nebo dokonce i poruchám celého systému. Pro detekci a eliminaci těchto defektů je možné použít některé z metod bojujících proti poruchám (angl. *fault-tolerant methods*), např. formální verifikaci nebo návrh systému na základě ověřených návrhových vzorů

(angl. *design patterns*).

Připomeňme si také některé katastrofy z oblasti *safety-critical* systémů. Prvním záznamem o možné katastrofě, která se naštěstí neudála, je chyba implementace programu s RTOS do vesmírného vozítka *Mars Pathfinder*. Operační systém se díky *watchdog* mechanismu resetoval, protože nastával problém známý jako *priority inversion*. Technicky se jedná o blokaci běhu vlákna s vyšší prioritou vláknem s nižší prioritou, který uzamkl sdílený objekt. Tato situace nastává pouze občasně, takže nebyla včas ošetřena v průběhu vývoje a testování [70]. Z této události ale vyplývá, že i v důkladně otestovaném systému opatřeném certifikáty mohou být skryté defekty na softwarové úrovni, které za určitých podmínek nebo v důsledku specifického sledu událostí mohou vést až ke katastrofě.

Dále můžeme jmenovat případ ze světa automobilismu. Například nečekané zrychlování aut značky Toyota (2013) při vymáčklém plynovém pedálu. Po důkladné analýze bylo zjištěno, že výrobce auta nepostupoval při návrhu aplikace pro RTOS v souladu s platnými doporučenými postupy danými standardem ISO 26262, a proto jediná událost změny bitu vlivem letící částice (angl. *Single Event Upset*) mohla způsobit tuto poruchu. Taktéž nesprávná implementace zásobníku (angl. *stack*) RTOS měla stejný následek. Další příčinou může být nesprávná implementace *watchdog* mechanismu, který nereagoval na defektem ukončené důležité vlákno (*task*) aplikace [74].

Potenciální katastrofou u *safety-critical* systémů je z hlediska RTOS hlavně zmeškání *deadline*, protože v tomto okamžiku jsou data vstupující do daného vlákna považována za neplatná, dokonce za potenciálně nebezpečná. Příčiny této chyby mohou plynout z různých softwarových, dokonce i hardwarových defektů. [66] udává, že až 21 % chyb v systému jako celku způsobí chybu v software, přičemž až z 87 % se jedná o chyby plánování úloh a real-time vlastností systému. Až 44 % chyb zůstane bez aktuálního efektu a 35 % způsobí okamžité selhání systému.

V systému tedy může potenciálně vzniknout mnoho rozličných defektů. Proto je hledání metod a technik pro včasnou detekci a zabránění nežádoucích stavů stále aktuální. Dokazuje to i nepřeberné množství literárních pramenů (článků, disertačních prací či technických zpráv projektů) a tematicky zaměřených konferencí, např. [7], [29] nebo [76].

Ve své práci se zaměřím na online formální verifikaci běhu softwarové části embedded systému s aplikacemi běžícími nad RTOS. Pokusím se také o spojení této techniky s teorií návrhových vzorů, což by mohlo vyústit v synergický efekt těchto *fault-tolerant* technik. Obzvláště pokud uvážím, že v *safety-critical* oblasti se už i v programátorské praxi s výhodou používá metoda návrhu systému založená na jeho modelu (angl. *Model-Based Design*).

1 CÍLE PRÁCE

V této kapitole shrnu předmět mé práce nejdříve v obecné formě vyplývající z výzev v dané oblasti, z čehož pak specifikuji předmět práce konkrétněji. Dále nastíním oblasti a možnosti očekávaného využití výsledků této práce. Nakonec stanovím hypotézy vyplývající z uvedeného předmětu této práce.

1.1 Předmět práce

Na začátku jsem měl zadání „HW implementace podpůrných algoritmů jádra RTOS s prvky funkční bezpečnosti“, což tvoří obecný rámec výzkumu. Takže obecným předmětem práce je přinést zlepšení (techniku či metodiku) v oblasti funkční bezpečnosti embedded systémů, tedy systémů s komplexnějším software, resp. se software na bázi operačního systému reálného času. Bez uvedení literatury obecné rešerše se akademický pohled i pohled z praxe shoduje na existenci následujících problémů a výzev v této oblasti:

- detekce a eliminace anomálií přístupu ke *cache* paměti,
- detekce a eliminace anomálií multi-vláknových aplikací (*deadlock*, *livelock*, *starvation* nebo *race condition*),
- detekce a eliminace chyb vykonávání systémových funkcí softwarového prostředí,
- detekce a maskování chyb přepnutí hradla (angl. *bit-flip error* či SEU) registrů nebo v paměťovém modulu,
- výpočet WCET a odhad výpočetní zátěže vláken aplikace,
- testování správné implementace synchronizačních elementů,
- identifikace a testování hazardních stavů aplikace (formální verifikace),
- testování softwarového vybavení přímo za běhu a detekce anomálie vedoucí k chybám,
- testování správné funkce hardware, na kterém běží softwarové vybavení,
- kvantitativní vyhodnocení pravděpodobnosti poruch v softwarovém vybavení,
- kvantitativní vyhodnocení zlepšení funkční bezpečnosti za použití dané techniky,
- definice vzorů návrhu částí softwarového vybavení, jejich implementace a možnosti integrace,
- nalezení vhodného matematického aparátu pro řešení problému exploze stavů (angl. *state explosion*) při testování a verifikaci softwarového vybavení,
- klasifikace poruchy a následného napravení v reálném čase,
- detekce, eliminace a maskování anomálií synchronizace a přístupu ke *cache* paměti v multi-procesorových systémech,
- nalezení vhodné metody pro automatizované generování bezpečného kódu.

Z těchto výzev jsem si jako předmět práce vybral **verifikaci softwarového vybavení za jeho běhu**, a to **na základě modelu sestaveného z návrhových vzorů**. Tím jsem se omezil na **detekci** defektů a chyb. Očekávám, že se výsledky práce promítnou i v jiných výzvách, jako je např. **detekce anomálií multi-vláknových aplikací**, **formální verifikace**, **detekce chyb vykonání systémových funkcí** a **definice návrhových vzorů částí softwarového vybavení**. Oproti off-line technikám statistické analýzy software tedy nebudu řešit problém exploze stavů, ale spíše problém detekce rozličných druhů chyb a potlačení falešných alarmů. Jelikož jsem se omezil na vestavné systémy obsahující operační systém reálného času, zaměřím se spíše na chyby plynoucí z jeho provozu, do kterých se promítají i chyby plynoucí z provozu aplikací.

1.2 Očekávané využití dosažených výsledků

Primární oblast využití výsledků této disertační práce je softwarové vybavení programovatelných prostředků průmyslové automatizace (např. CNC řízení, řízení robotického manipulátoru, bezpečnostní programovatelné komponenty nebo softwarové vybavení PLC), kde jsou kladeny vyšší požadavky na funkční bezpečnost (např. SIL3 dle standardu IEC61508). Je však zřejmé, že techniky, které jsou předmětem této práce, mohou být použity i v jiných oblastech, např. procesní automatizace, automobilismus, medicínská technika, vojenská technika, nebo dokonce letecká technika. Nutno podotknout, že cesta od vývoje bezpečnostních algoritmů až po jejich integraci do produktů je dlouhá a není předmětem této práce, takže TRL3 podle European Space Agency by byla nad očekávání dosažená úroveň.

Výzkum a vývoj techniky formální verifikace za běhu aplikace nabízí využití hlavně v akademickém prostředí, a to výzkum a vývoj sofistikovaných technik maskujících a eliminujících chyby, jako je např.:

- dynamická rekonfigurace CPU / vlákna na jiné místo v hradlovém poli,
- klasifikace adekvátní akce kontrolního bezpečnostního modulu systému (FCCU),
- detekce jádra s chybou v *lock-step* / *hot-swap* architektuře,
- vyhodnocení degradace a provozuschopnosti systému,
- adekvátní reakce na pravděpodobné nesplnění *deadline* vlákna,
- testování algoritmů umělé inteligence jako dohledového systému.

V praxi by výsledky mohly vést k integraci subsystému verifikujícího běh operačního systému s aplikacemi, což vede z hlediska funkční bezpečnosti zařízení ke zvýšení **diagnostického pokrytí**, a tím k vyšší kvantifikaci bezpečnosti danou příslušnými normami. Dalším důsledkem pro praxi může být:

- použití návrhových vzorů při vytváření designu softwarového vybavení,

- automatické generování kódu aplikace a testovacího subsystému na základě formálního modelu či popisu v programovacím jazyce (např. C real-time extension [101]),
- podnícení lepší implementace a integrace metod bojujících proti chybám.

1.3 Stanovení hypotéz

Zde stanovím hypotézy vyplývající z uvedeného předmětu, které se v průběhu práce pokusím potvrdit, nebo vyvrátit: 1) Kontrolní subsystém běhu RTOS programu zlepší diagnostické pokrytí systému. 2) Integrací návrhových vzorů do procesu vývoje systému je možné zlepšit spolehlivost (funkční bezpečnost) celého systému. 3) Pro popis procesorového software za účelem monitorování je možné použít formální nástroje určené pro modelování takovýchto programů. 4) Algoritmy funkční bezpečnosti je možné s výhodou implementovat i v hardware.

1.4 Členění dokumentu

Po stanovení předmětu práce a výchozí hypotéze (viz výše) uvedu stručně teorii operačních systémů reálného času (viz kap. 2.1) a funkční bezpečnosti (viz kap. 2.2). Dále provedu literární rešerši (viz kap. 3) inovátorských přístupů v oblasti kontroly běhu systému a uvedu literární díla, ze kterých moje práce vychází. Provedením rešerší na téma formální metody (viz kap. 4), možnosti hardwarové implementace algoritmů (viz kap. 5) a návrhové vzory (viz kap. 6) dostanu solidní základ poznatků, který spolu s měřením scénářů obsahujících rozličné chyby (viz kap. 7) použiji jako výchozí sadu poznatků, který umožní definovat kontrolní subsystém běhu aplikace (viz kap. 8) a implementovat jej za účelem prvotního ověření funkčnosti (viz kap. 9).

2 TEORETICKÝ ÚVOD

V této kapitole popíši relevantní teoretické základy a poznatky, na kterých budu stavět v dalších kapitolách. Cílem teoretického úvodu není popsat vyčerpávajícím způsobem současné informace a znalosti, ale uchopit současný stav poznání relevantně k tématu práce. Tedy jsou zde shrnuty znalosti, nástroje a definice uznávané akademickou obcí i programátorskou většinou. Nejdříve krátce prezentuji teorii operačních systémů reálného času (viz kap. 2.1) a pak udělám výtah z méně probádané oblasti, a to funkční bezpečnosti (viz kap. 2.2), resp. standardu IEC61508. Jelikož se jedná o fakta, která jsou interpretována všemi autory literárních pramenů téměř stejně, omezím rozsah a zmíním pouze fakta a východiska, která jsou stěžejní pro mou práci.

2.1 Operační systémy reálného času

Operační systémy reálného času (RTOS) jsou zvláštní podskupinou operačních systémů, které mají zvýšené nároky na provedení jednotlivých úloh systému v závislosti na čase. Většinou jsou tyto systémy nasazovány do vestavných zařízení. V této kapitole je rozebráno, jak tyto systémy fungují, jak se dělí, jaké mají vlastnosti a proč se používají. Informace pro tuto kapitolu čerpám hlavně z literatury [77], [81] a [126]. Z hlediska metod boje proti chybám slouží jako příklad [94], [116], [138] a analýzou těchto problémů se zabývá např. [63] a [23].

2.1.1 Definice

Definice operačního systému reálného času má více formulací v závislosti na úhlu pohledu. Z obecného pohledu je platná definice: „Operační systém reálného času je takový operační systém, který je schopen provádět výpočty a reagovat na události v předem definovaných časových intervalech (angl. *deadline*)." [77] Definice také může vycházet z teorie informací, a to, že operační systém reálného času je takový operační systém, který zaručuje, že jsou informace v systému časově a místně konzistentní. Důležitým parametrem tedy oproti běžným operačním systémům není průměrná doba vykonání reakcí, ale doba reakce v tom nejhorším možném případě [106].

2.1.2 Dělení

Soft real-time vlákna jsou určená pozvolným tvarem *funkce užitečnosti* a stačí u nich, když budou vykonány v daném průměrném časovém intervalu nebo do daného průměrného času. S pozdě příchozími daty může ještě úloha s jistými omezujícími faktory pracovat.

Firm real-time vlákna jsou určena strmým klesajícím tvarem funkce užitečnosti a musí být vykonána v definovaném intervalu, příp. s minimálním zpožděním. Pokud by se nestihla vykonat, došlo by ke snížení kvality funkce či její poruše.

Hard real-time vlákna jsou určena nespojitým tvarem funkce užitečnosti a musí být bezpodmínečně vykonána v definovaném intervalu. Pokud by se nestihla vykonat, došlo by k selhání systému vlivem této poruchy se všemi důsledky (finanční ztráta, zranění). Data, která přijdou pozdě, jsou bezcenná, neplatná, či dokonce potenciálně nebezpečná. Úloha tedy s nimi dále nesmí pracovat.

Dále můžeme RTOS dělit dle implementace na **micro-kernel**, který odděluje dílčí funkce, a **monolythic kernel**, které se chová jako zapouzdřené jádro. Dle rozhraní můžeme kategorizovat POSIX systémy od ostatních, kde výrobce definuje názvy a atributy systémových funkcí. Dle typu plánovače úloh se RTOS dělí na časem řízené (angl. *time driven*), událostmi řízené (angl. *event driven*) a smíšené.

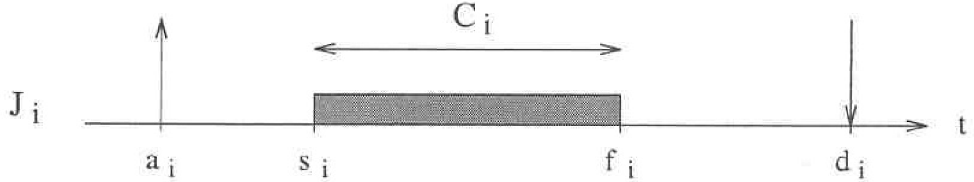
2.1.3 Popis

Operační systém reálného času je v podstatě operační systém, který má některé své části (komponenty) upraveny tak, aby systém jako celek byl časově deterministický. Mezi hlavní komponenty RTOS patří deterministický plánovač vláken, prostředky pro meziprocesovou komunikaci, prostředky pro správu paměti, systém pro správu reálného času a popř. ovladače zpřístupňující hardware do vyšších vrstev (HAL).

Plánovač se stará o včasné a správné přepnutí kontextu na jiné vlákno (systém je tedy preemptivní). Jeho determinismus spočívá v definovaném výběru vlákna dle daných pravidel, jako je priorita nebo stav vlákna ve smyslu synchronizace s ostatními vlákny, a v definovaném maximálním časovém intervalu, do kterého musí celý cyklus svých operací (prohledat a vybrat vhodné vlákno k vykonávání, uložit kontext předchozího vlákna, načíst kontext dalšího vlákna) provést. Plánovač tedy přepíná vlákna dle daného algoritmu (např. RMS, EDF či FCFS round-robin).

Meziprocesorová komunikace je řešena na základě sdílené paměti nebo systémem předávání zpráv. Chybou je, když proces neoprávněně přistupuje do paměti jiného procesu. Přístup do paměti vlastního procesu je zabezpečen součinností operačního systému a MMU jednotkou, resp. MPU jednotkou. Vyšší stupeň eliminace chyb a zároveň rychlejší dobu práce s pamětí poskytuje systém s hardwarově oddělenou pamětí pro každý proces, což na druhou stranu vede ke složitějšímu mechanismu při potřebě sdílené paměti.

Každá úloha má své parametry (viz obr. 2.1), pomocí nichž jsou úlohy řazeny plánovačem. Každá úloha J_i je vykonána pomocí vlákna za dobu C_i. Úloha mohla být zařazena od doby a_i a musela se vykonat do deadlinu d_i, přičemž reálně se vykonala od doby s_i do doby f_i.



Obr. 2.1: Parametry úlohy [22]

2.1.4 Chyby v RTOS

U striktních hard real-time systémů musí být řazení úloh plně deterministické. Tyto úlohy musí být dopředu známé a nesmí dynamicky vznikat neočekávané úlohy. U těchto systémů jsou úlohy většinou naplánovány staticky offline a je provedena časová analýza využití procesorového času. Spolehlivost těchto systémů je dostatečně zajištěna pomocí standardních metod boje proti chybám.

U systémů s periodickými a aperiodickými úlohami může dojít vlivem chyb, způsobených přetížením, k nestandardnímu chování jádra, hardwarovým projevům nebo k poruše plánování úloh systému. Tyto poruchy, jako je nestihnutí *deadline*, *deadlock*, *livelock* a *race condition*, jsou při použití dosavadních metod detekovány, až nastanou. Systém v drtivé většině reaguje pouze akcí reset. Systém se ale skládá z úloh s různými prioritami, takže je zřejmé, že pro zajištění chodu méně důležitých úloh nemusí být systém hned restartován, ale např. rekonfigurován se zajištěním chodu důležitých úloh. Tímto se zvýší provozuschopnost systému jako celku, i když dojde k degradaci jeho funkcí.

Pomocí speciálních nástrojů je možné modelovat a analyzovat plánování úloh dle jejich parametrů. Z toho je možné zjistit, zda všechny úlohy teoreticky stihnou své deadliny. U úloh, které nemají pevně danou dobu vykonání, se počítá s nejhorší možnou dobou nebo s efektivní dobou. Pokud v systému navíc mohou dynamicky vznikat periodické či aperiodické úlohy, je třeba provést statistickou analýzu (více v [25]) nebo provést test zařaditelnosti úlohy do systému početně (více v [113]).

Za chybu plánovače je považováno:

- nesprávně vybraná úloha k vykonávání,
- překročení deadline úlohy,
- překročen limit času přepnutí kontextu,
- přepnutí kontextu v nesprávný čas,
- *deadlock* - úlohy se navzájem blokují ve vykonávání synchronizačními mechanismy,
- nekonečná smyčka,

- *livelock* – zacyklení částí úloh díky špatné synchronizaci,
- *starvation* – znemožnění vykonání úloh kvůli nekonečnému upřednostňování jiných úloh nebo livelocku,
- *race condition* – nesprávný souběh úloh, plánování je nepředvítalné a špatně načasované.

Pokud CPU musí vykonávat pro detekci chyby fault-tolerant či bezpečnostní funkce, musí mít tyto funkce vyšší prioritu a nesmí být ovlivněny vykonáváním ostatních úloh. Problém nastává, pokud tyto funkce vyžadují zdroje používané v jiných úlohách. Tento problém musí vyřešit plánovač systému. Pokud ale systém dokáže splnit všechny uvedené případy, může mít přece jenom problémy ve ztížených podmínkách, jako je zvýšená výpočetní zátěž či výskyt mnoha přechodných chyb. Více je uvedeno v [91].

2.1.5 Více-procesorové systémy

Výzkum v oblasti více-procesorových (angl. *multi-core*) systémů prudce stoupl, jelikož poptávka v komerční sféře stále roste. Mezi hlavní důvody stále častějšího nasazování patří skutečné paralelní vykonávání úloh, vyšší produktivita, dobrá dostupnost a možnost implementace algoritmů pro detekci a eliminaci chyb, jako je prostorová redundance (angl. *spatial redundancy*) nebo provozní testování (angl. *built-in tests*).

V těchto systémech může RTOS stejně jako obecný OS tvořit *symetrickou*, *asymetrickou*, nebo *non-uniform* NUMA architekturu. Navíc ale musí zajistit časově deterministickou odezvu meziprocesových synchronizačních mechanismů. Tím je obvykle tato odezva delší oproti běžným OS.

Úprava RTOS pro více-procesorové systémy má podobný základ jako u standardních OS, avšak skýtá daleko více problémů týkajících se zaručení *determinismu*, plánování úloh (viz [79]), anomálií mezi-procesové komunikace a anomálií vznikajících v důsledku společné *cache* paměti. Příklad procesu portování FreeRTOS na multi-procesorovou platformu uvádí [93].

2.1.6 Vlastnosti

Požadavek na nasazení víceúlohového operačního systému nemusí pramenit pouze z požadavku na programátorský komfort, kdy dekompozice úloh na jednotlivá vlákna vede ke snadnější modifikaci a ověření, ale také z požadavku na zvýšenou odolnost systému proti poruchám, kdy selhání jednoho vlákna nemusí nutně vést k selhání celého systému. Další výhodou je, že lze k jednotlivým úlohám přiřadit prioritu, a tím může být ovlivněna jejich šance na vykonání plánovačem.

Portovatelnost na různé procesorové architektury je umožněna vyčleněním HAL vrstvy a dále kultivovaností kódu, kterým jsou jednotlivé části RTOS tvořeny. Jádro RTOS, resp. plánovač úloh, tedy nemusí být závislé na hardwarovém vybavení.

Splnění podmínky definice RTOS nevychází pouze z jeho správné implementace, ale i ze správné implementace jednotlivých úloh a jejich integrace pomocí meziprocesorových mechanismů a komunikace.

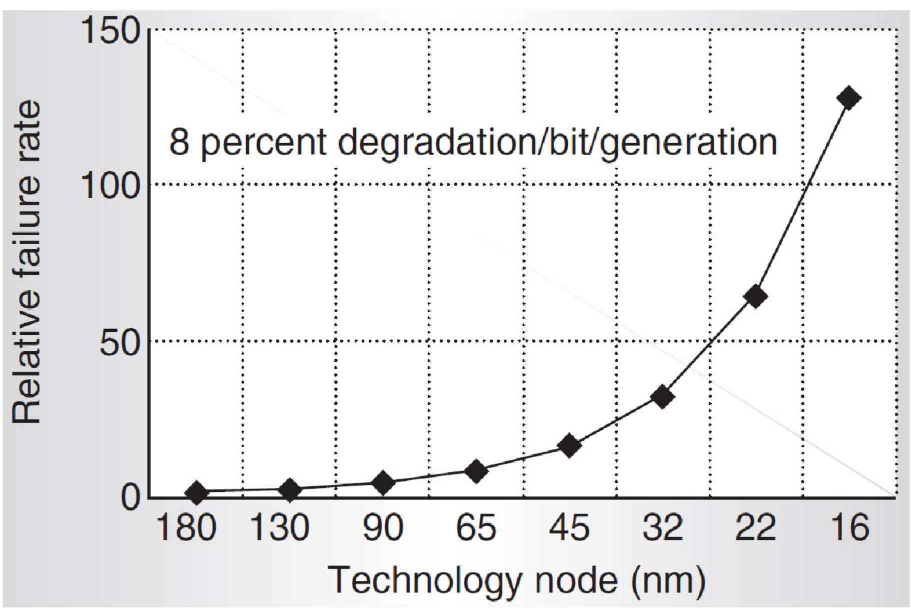
2.2 Funkční bezpečnost

Funkční bezpečnost je nedílnou součástí celkové bezpečnosti výrobku či systému. Zařízení musí správně reagovat na vstupy systému, včetně pravděpodobných chyb operátora, selhání hardwaru nebo softwaru a změn prostředí. Se vzrůstající komplexitou zařízení a systémů stoupá riziko výskytu chyby (viz obr. 2.2), přičemž výrobci i provozovatelé musí mít jistotu, že tyto systémy spolehlivě fungují s maximální efektivitou a zároveň jsou bezpečné. Zavedením konceptu rizika lze toto riziko kvantifikovat jako součin pravděpodobnosti události a jejích následků.

V této kapitole jsou nejdříve vymezeny pojmy, jako je spolehlivost, chyba a selhání. Dále budou analyzovány nejčastější zdroje chyb systému a jejich možné projevy. Poté budou rozebrány standardní metody používané v boji proti chybám. Nakonec budou shrnuty relevantní části normy IEC 61508 a požadavky pro certifikaci na SIL3, což je nejvyšší možná úroveň bezpečnosti dosažitelná při použití software v daném certifikovaném zařízení. Proces certifikace na SIL 3 je časově a finančně nákladný proces, který je určený primárně pro výrobce zařízení či systémů a ne pro akademické účely, proto není tento proces předmětem této práce.

2.2.1 Definice pojmů

Funkční bezpečnost – metodika pro návrh, implementaci, verifikaci a údržbu automatických bezpečnostních funkcí systému (SIF), aby vykonávaly svou funkci správně nebo aby uvedly systém do definovaného stavu. Definuje tedy všechny aspekty výše zmíněných procesů, nástroje, jejichž použití vede k deklaraci, a dokonce kvantifikaci bezpečnostních problémů systému, a také definuje metriku pro posuzování míry funkční bezpečnosti (SIL). Funkční bezpečnost jako vlastnost systému je blíže k průmyslovým zařízením specifikováným normou ČSN ISO 12100, která popisuje proces analýzy a posouzení, a dále normou ČSN ISO 13849-1, která mimo jiné definuje požadavky na spolehlivost, odolnost proti poruchám, diagnostické pokrytí, imunitu proti poruchám se společnou příčinou, integritu systému a také řízení procesů funkční bezpečnosti.



Obr. 2.2: Závislost četnosti přechodných chyb na použité technologii [16]

Spolehlivost - (angl. *reliability*) je obecná vlastnost objektu spočívající ve schopnosti plnit požadované funkce při zachování hodnot stanovených provozních ukazatelů v daných mezích a v čase podle stanovených technických podmínek [65]. Hlavní ukazatel se udává jako hodnota MTBF, kterou obvykle stanoví výrobci jednotlivých komponent. Řetězením subsystémů lze spočítat tuto hodnotu pro celý plně definovaný systém. U složitějších systémů je možné použít pro analýzu spolehlivosti různých nástrojů, např. teorii Markovových procesů, která se zabývá stavovou analýzou stochastických systémů bez paměti v reálném čase. Jedná se tedy o kvantitativní část funkční bezpečnosti. Vztah mezi MTBF a spolehlivostí je uveden v 2.1.

$$R(t) = e^{-\frac{t}{MTBF}} = e^{-t\lambda} \tag{2.1}$$

Chyba – ukončení schopnosti zařízení vykonávat požadovanou funkci. Chybu zapříčiňuje nějaký **defekt**. Pokud se chyba neřeší, může způsobit **poruchu** systému jako celku nebo dále vyústit v **selhání** systému či **katasrofu**.

Přípustné riziko – riziko, které je přijatelné v daných souvislostech založených na běžných hodnotách společnosti.

2.2.2 Zdroje chyb

V této sekci se zaměřím na zdroje chyb v doméně vestavných systémů, resp. integrovaných obvodů. Takovéto chyby se dělí na:

- Systematické – chyby, které vznikají ve fázi návrhu a implementace, tedy mohou být podchyceny (např. errata, design).
- Nahodilé – chyby, které se objeví v průběhu činnosti systému v nepředvídatelném čase a které mohou být pouze modelovány pomocí teorie pravděpodobnosti.
  - Vážné – poškození součástek (např. zkrat nebo mechanické přerušení)
  - Přechodné – změna stavu hradla (SEU) způsobená částicemi nebo EMI

Chyby v RTOS aplikaci propagují často z hardwarových defektů (*soft-errors*). [66] udává, že až 21 % defektů zpropaguje jako chyba v software, přičemž až z 87 % se jedná o chyby plánování úloh a real-time vlastností systému. Až 44 % chyb údajně chod systému neovlivní a 36 % způsobí okamžité selhání systému, což se projeví jako neplatná informace nebo jako špatně vykonaná instrukce. Některé chyby nemusí vést nutně k selhání systému a systém tedy může dále plnit svoji funkci. Tyto chyby se špatně hledají, protože se většinou projeví později za jiných okolností [51]. Může se také jednat o chyby se společnou příčinou (angl. *common cause faults*), které postihnou redundantní části stejně (např. porucha napájení nebo zdroje hodinového signálu).

Míru rizika plynoucí z potenciálních chyb lze vyjádřit pomocí metod uvedených v IEC 61508. Mezi nejčastější ukazatele patří pravděpodobnost výskytu poruchy za hodinu / při přístupu (angl. *Probability of Failure per Hour / Demand*), bezpečné zbytkové riziko (angl *Safe Failure Fraction*) či kvantitativní priorita rizika (angl. *Risk Priority Number*). K určení těchto ukazatelů jsou doporučovány metody uvedené v IEC 61508, a to stromový diagram chyb (angl. *fault tree diagram*), tabulka příčin s následky (angl. *Failure modes, effects, and diagnostic analysis*), vyčíslení ztráty produkce nebo statistických údajů o četnosti výskytu.

2.2.3 Metody boje proti chybám

Systémy odolné proti poruchám (angl *Fault-tolerant systems*) používají různé metody boje proti chybám (angl *fault-mitigation method*) a jejich kombinace pro omezení chyb a jejich proměn v poruchy systému. Obecně se dělí do kategorií na metody:

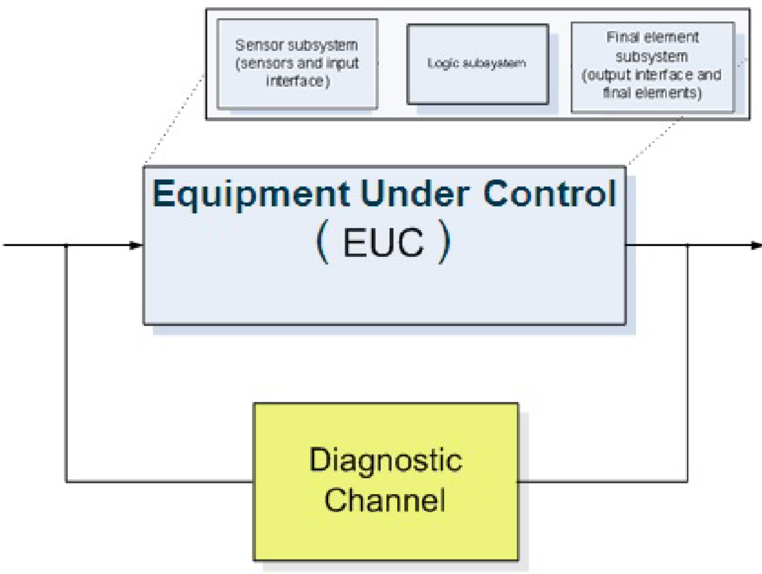
- předcházení chybám (angl. *fault avoidance*) – snaha o eliminaci chyby dříve, než vznikne,
- detekce chyb (angl. *fault detection*) – včasné odhalení chyby s minimalizací planých poplachů,
- maskování chyb (angl. *fault masking*) – úprava systému tak, aby byl schopen správně plně nebo částečně nadále fungovat i za přítomnosti chyby.

V dalších podkapitolách projdu metody doporučované normou IEC 61508 a nejčastěji používané techniky v programátorské praxi. Tento výčet není zdaleka úplný. Na druhé straně z analýzy [129] vyplývá, že i když systém disponuje různými technikami,

mohou i přesto některé chyby zůstat nedetekovány a zpropagovat až do uživatelské aplikace nebo zapříčinit pád systému.

Monitorování

Řídicí subsystém je monitorován subsystémem (viz obr. 2.3), a to jako black box, kdy monitorovací subsystém má k dispozici pouze vstupy a výstupy řídicího subsystému, nebo jako white box, kdy monitorovací subsystém má k dispozici port pro sledování vnitřních stavů řídicího subsystému. Tento způsob detekuje určité procento chyb, a to v závislosti na rozhraní mezi monitorovacím a monitorovaným zařízením.



Obr. 2.3: Schéma systému s EUC [85]

Lockstep

Tato technika využívá principu stejného chování dvou identických CPU jednotek. Výstupní signály, popř. vnitřní registry dvou paralelně běžících CPU jednotek, jsou po každém hodinovém taktu porovnávány externím obvodem, který nezatěžuje chod CPU. Pro detekci odchylky hodinového signálu je hodinový signál jednoho z CPU fázově posunut. Tato metoda tedy odchytí chyby, které způsobí odlišný chod jedné z CPU jednotek způsobený přechodnými chybami. Nevýhodou je, že není možné určit, která z CPU byla zastižena chybou, a proto se musí restartovat celý systém, popř. zaktivnit jiný procesor. Tato technika také neumí detekovat chyby RTOS způsobené nesprávným návrhem nebo implementací aplikace.

Některé modifikace této techniky umožňují vrátit systém do předchozího stavu před chybou (viz [107]) nebo vylepšují tuto techniku o detektor vadného procesorového kanálu s následnou nápravou (viz [103] a [55]). Další řešení umí přeprogramovat sledované CPU na jiné místo v hradlovém poli (viz [144] a [2]). Jedná se ale o akademická řešení.

Pro detekci jádra obsahujícího chybu se může použít RESO metoda, která spočívá v porovnání normálního výstupu a bitově posunutého výstupu vytvořeného z bitově posunutého vstupu. Tato metoda je ale vhodná pouze pro real-time logiku, jako je např. ALU jednotka procesoru. Také samotná rekonfigurace se může zrychlit rekonfigurováním pouze části (angl. *framu*) s chybou, pokud je tato metoda nasazena v hradlovém poli. *Frame* s chybou se detekuje pomocí CRC kontroly a rekonfiguruje pomocí dalšího malého soft-core. Tento přístup je popsán v [114].

Modulární systémy

Aby bylo možné určit, která CPU zaznamenala chybu, je nutné použít alespoň tři paralelní CPU. Jedná se o tzv. TMR systém, který se skládá ze tří řídicích komponent a vyhodnocovacího subsystému (angl. *voteru*), který rozhoduje o výsledném signálu na základě tří příchozích signálů. *Voter* rozhoduje např. na základě několikastupňového mediánu nebo pomocí Kalmanova filtru.

Testování

Testování komponent systému si vyžadují specifikace vyšších úrovní SIL. Většina testovacích rutin se spouští při startu systému. Pokud systém musí běžet dlouhou dobu bez vypnutí, je vhodné vyvolat testování v pravidelných intervalech a zajistit tím lepší diagnostické pokrytí. Testovány mohou být hardwarové periferie systému, informace s výsledky software nebo vlastní chod softwaru, např. pomocí checkpointů. Tato metoda zjistí chyby, které vznikly v testované části hardware a které znemožňují jeho správnou funkci [145].

Testování pomocí kontrolních součtů (CRC) má za úkol zajistit integritu dat. Tato technika je využívána hlavně při komunikaci a pro detekci chyb paměťových úložišť. Některé pokročilé techniky umí i opravit chybu jednoho *bit-flip* pomocí přidaného paritního bitu (ECC), nebo dokonce i více bitů (viz Leon4FT processor).

Testování v pravidelných intervalech může monitorovat např. vytížení procesoru pro detekci stavů zamrznutí systému způsobených nekonečnými smyčkami, abnormálně dlouhým čekáním, vytížením jádra OS, prodloužením času operací s pamětí pro detekci defektů komponent nebo prodloužením času zápisu do mapovaných registrů, jako je např. GPIO. Při tomto způsobu je nutné obezřetně zvolit frekvenci

a čas testování, aby nebyl testovaný systém příliš zatížen a aby se chyby detekovaly dříve, než se projeví jako selhání systému. Více o této metodě je uvedeno v [149].

Teorie testování softwarových aplikací nabízí mnoho přístupů a technik, např.:

- funkcionalita – testování správné funkcionality subsystémů dle požadavků,
- kvalita – měření výkonnosti a stability,
- data – testování dat v daném rozsahu, náhodně nebo jen hraniční hodnoty,
- pokrytí – průchod všemi možnými stavy (např. *code coverage* a MC/DC *coverage*),
- stres – testování reakcí na neočekávané hodnoty,
- chyby – testování správné reakce algoritmů na injekci chyb.

Návrh systému

Návrh systému je jedna ze stěžejních oblastí pro eliminaci chyb. Správným návrhem mohou být odstraněny chyby způsobené EMI, teplotou a jinými vnějšími vlivy. Může tedy být eliminována většina chyb, jejichž zdrojem je nevhodný hardware či software.

Metoda návrhu systému z modelu (angl. *model-based design*) pomáhá eliminovat chyby již při návrhu, obzvláště v kombinaci s komplexním testováním modelu a případně následným generováním kódu z modelu. Uplatnit se může hlavně formální verifikace modelu za účelem záruky požadovaných vlastností. Tato oblast začíná být v praxi hojně využívaná, což je žádoucí i z požadavků standardu IEC 61508. V akamedickém prostředí existuje nepřeberné množství literárních pramenů, které se oproti komerčnímu prostředí zabývají touto oblastí komplexně, např. [151] nebo [88].

Systém nutně nemusí být vytvořen přímo z modelu, ale model může být vytvořen a použit pouze pro offline verifikaci. Pro verifikaci může být jako formální model použit např. Markovův řetězec v diskrétním čase (viz [118]) nebo nástroj LTL (viz [47]) pro definici vyhodnocovaných vlastností.

Watchdog

Watchdog se jako nástroj pro detekci přechodných chyb používá v praxi nejčastěji, protože standard IEC 61508 jeho nasazení vyžaduje. Na druhé straně to mnohdy výrobcům stačí a ani se nezamýšlí nad tím, že existují různé architektury implementace. Tím jsou v základu kombinace hardware a software implementace [115]. Dále procento detekovaných chyb závisí na implementaci v software, obzvláště v RTOS systémech [94]. Pro systémy odolné proti poruchám se v hardware implementaci také může využít kaskádní architektura, při které se postupně aktivují jednotlivé úrovně bezpečnostních funkcí (např. restartovat vlákno, spustit měkký reset, restartovat systém). Softwarová implementace umožňuje šetřit hardwarové prostředky (na úkor snížení bezpečnosti) a tvořit komplexnější struktury [98].

Nevýhodou je provázanost se systémovým časovačem, tedy jeho odchylky tvoří systémovou chybu. Tento problém částečně řeší hardwarová implementace. Problémem také je určit správnou vybavovací hodnotu a začlenit správně obnovovací sekvenci do systému. Vzniká tedy riziko pozdního vybavení.

Ačkoliv je watchdog nejjednodušší a přitom efektivní technika, vznikají na toto téma nové výzkumy a technické zprávy, které se snaží tento nástroj zdokonalit, např. [104].

Redundance

Redundance obecně může být implementována v různých doménách, a to **místní**, **časové** a **datové**. Místní redundance spočívá v nasazení n-modulárních systémů (viz kap. 2.2.3). Časová redundance se může použít v případě, kdy zbývá výpočetní čas po vykonání všech vláken v dané periodě při normálním provozu a kdy se mohou tyto výpočty provést po chvíli ještě jednou pro porovnání shody s předchozími (viz [12] a [95]). Vychází se tedy z předpokladu, že přechodné chyby nebudou působit v různém čase stejně. Časová redundance a zároveň redundance místa může být využita pro vytvoření distribuovaného řídicího systému maskující chyby, a to replikací nebo zařazením úlohy do fronty ještě jednou (více v [68]). Datová redundance multiplikuje data v systému, která se tak můžou obnovit v případě zjištění chyby.

V průmyslové automatizaci se tato technika (*hot-swapping*) hojně využívá na úrovni hardware (sběrnice, snímače) a na úrovni PLC. Jedná se o metodu, kdy se za běhu systému přepojí elektricky část systému, aby byl zajištěn jeho chod i při poruše. Tento systém tedy potřebuje detektor poruchy v chráněném subsystému. Samotné přepojení také určitou dobu trvá, ale např. u nejvyšší řady PLC firmy Schneider lze dosáhnout přepnutí v rámci jednoho cyklu PLC. Připojený subsystém musí začít vykonávat program tam, kde předchozí skončil, což lze u PLC dobře zajistit. Redundance tedy umožňuje kromě zvýšení funkční bezpečnosti zvýšit i dostupnost (angl. *availability*) daného systému [43].

Offline analýza

Matematické metody offline analýzy jsou stále předmětem aktivního výzkumu, i když se jedná o dlouhodobě bádanou oblast. Základem pro tyto techniky je mít věrohodný model systému vyjádřený pomocí některého z formálních jazyků. V množství publikací, jako je např. [9], lze nalézt vhodný přístup pro verifikaci dané funkce modelu (angl. *model-checking*). Jedná se o metody testování hypotéz (angl. *theorem proving*), formální matematické metody (např. [33]), statistické metody (např. [90]), nástroje umělé inteligence (např. [78]) a hrubé techniky procházení stavového prostoru

pomocí simulací (např. [117]). Metody statické analýzy však většinou trpí společným problémem, a to nepokrytím celého stavového prostoru (angl. *state explosion problem*).

Runtime verifikace

Oproti offline analýze nabízí verifikace za běhu řešení problému stavové exploze testováním pouze daného aktuálního stavu nebo cesty oproti požadavkům, které mohou být vyjádřeny jako konečné stavové automaty (např. [125]), regulární výrazy, pomocí syntaxe lineární temporální logiky (např. [136]) nebo jiných nástrojů vhodných pro implementaci do daného prostředí. Tímto odpadá náročný krok vytváření věrohodného modelu systému, ovšem za cenu snížení diagnostického pokrytí chyb. Nedostatečné diagnostické pokrytí zapříčiňuje stále aktivní výzkum nových technik a přístupů v této oblasti (např. [27] a [148]). Techniky offline a online verifikace se mohou také vzájemně doplňovat (viz [147]). Tato metoda může tedy sloužit pro testování funkční bezpečnosti, ale i jiných vlastností systému, jako je např. kybernetická bezpečnost, zjištění debuggovacích informací, validace a profilace.

Specializované RTOS

V aplikacích s velkým důrazem na funkční bezpečnost a spolehlivost se mohou s výhodou použít specializované varianty RTOS, které už prošly certifikací dle standardu IEC 61508. Tyto RTOS jsou např. SafeRTOS, který vychází z FreeRTOS, VxWorks, uC nebo Sciopta (více v [84]). Modifikace pro soulad s tímto standardem obecně zahrnují implementaci průběžného testování, alokaci proměnných staticky a přijmutí omezení plynoucích z MISRA C standardu. Další přístup spočívá v použití starších RTOS, které jsou odzkoušené časem, jako je RTEMS nebo MaRTE OS. Větší spolehlivosti, zejména snížení lidského faktoru při programování, je možné dosáhnout použitím časem odzkoušených programovacích jazyků pro real-time aplikace, jako je Ada a Pearl (viz [127]).

Sandboxing

Vytvořením virtuální vrstvy nebo rovnou celého stroje (angl. *hypervisor*), na kterém se bude cílová aplikace obsahující potenciální chyby spouštět, je možné docílit toho, že se sice díky chybě aplikace virtuální stroj ukončí, ale hostitelský stroj bude dále vykonávat řádně svoji funkci. Daný reálný stroj se tedy odizoluje od aplikace, a není tudíž úplně vystaven efektu vzniknuvších chyb. Tato metoda je však vhodná pro systémy s vyšším výpočetním výkonem.

2.2.4 **Standard** IEC 61508

V této podkapitole vypíši poznatky a postřehy ze standardu IEC 61508 (viz [30]) a rozšiřujících publikací (viz [120]) týkající se mé práce. Z tohoto standardu vychází jednotlivé standardy pro jednotlivé oblasti, jako je automobilismus, medicína, letectví a průmysl. V oblasti průmyslu se ale jedná o výchozí standard, který musí splňovat nová zařízení. Pro oblast funkční bezpečnosti ve strojírenském průmyslu derivuje tento standard do normy ČSN EN 12100 věnující se posouzení rizik a dále ČSN EN 13849-1 věnující se určení integrity bezpečnosti pro stroje. První část standardu se věnuje pojmům a metodám pro klasifikaci a kvantifikaci chyb. Druhá část se věnuje požadavkům a doporučením pro snížení rizika na hardwarovou část systému. Třetí část se věnuje požadavkům a doporučením pro snížení rizika na softwarovou část systému. Čtvrtá část obsahuje doplňující informace.

Hlavním cílem standardu IEC 61508 je poskytnout výchozí sadu nástrojů a definice pro analýzu a snížení zbytkového rizika ze všech zdrojů nebezpečí pod úroveň přípustného rizika, což je riziko přijatelné v daných společenských souvislostech. K tomu jsou definovány kategorie integrity bezpečnosti (SIL). Úrovně SIL jsou definovány dle rozsahu újmy, frekvence újmy a dle možností vyhnutí se nebezpečí při dané pravděpodobnosti poruch. Tyto čtyři úrovně také definují přípustnou pravděpodobnost chyby za hodinu nebo při přístupu. Aby systém mohl vykonávat bezpečnostní funkce pro minimalizaci škod a újmy na zdraví na dané úrovni, je nutné, aby bezpečnostní komponenty tohoto systému dosahovaly určité spolehlivosti. Paradoxem potom je, že se může uplatněním bezpečnostních prvků spolehlivost celého systému snížit. S vyšší bezpečností také vzrůstá pravděpodobnost tzv. falešných poplachů, což je dáno nastavením bezpečnostních prvků (viz [137]). Požadavek na zajištění bezpečnosti má ale vyšší prioritu.

Úrovně SIL 3 a SIL 4 obsahují nejpřísnější požadavky na zařízení. Pro software se jedná o integraci *watchdog*, použití semi-formálních nebo lépe formálních metod při návrhu, fyzické oddělení EUC od monitorovacího subsystému, vysoký stupeň integrace technik boje proti chybám s testovacími subsystémy (testování paměti, sběrnic aj.), použití nejlépe dlouhodobě odzkoušených komponent (testování v řádu 1–10 let), zvýšení procenta bezpečných chyb (angl. *Safe-Failure Fraction*), testování injektováním chyb alespoň na úrovni modulů, použití online či offline analýzy, omezení chyb se společnou příčinou (angl. *common cause failure*) a jiné minoritní požadavky. Norma taky poněkud vágně vyžaduje použití *state-of-the-art* technik.

3 LITERÁRNÍ ROZBOR

V této kapitole uvedu literární prameny (knihy, disertační práce a dlouhodobý výzkum), které jsem doposud objevil a které považuji za stěžejní či přínosné pro mou práci. Z těchto pramenů čerpám informace pro celou práci, snažím se k nim přistupovat s lehkým nadhledem a vybírat si nezpochybnitelné informace a myšlenky, které uvádějí i jiní autoři, např. v konferenčních příspěvcích. Literatura, kterou používám jen v dílčích kapitolách práce, je uvedena vždy na začátku kapitoly. Dále uvedu některá relevantní inovátorská řešení, která se ubírají podobným směrem jako moje práce nebo která považuji za přínosná v této oblasti.

3.1 Literární prameny

Disertační práce [5] obhájená na univerzitě RWTH Aachen v roce 2010 pojednává o použití pojetí návrhových vzorů v oblasti safety-critical embedded systémů. Autor definuje katalog relevantních návrhových vzorů. Dokonce je uvedena metodika pro kvantifikaci zlepšení spolehlivosti jednotlivých vzorů oproti základnímu systému, dále je uvedena míra doporučení použití jednotlivých vzorů dle normy IEC 61508, odhad nákladů pro implementaci vzoru a také dopady na jiné aspekty programu, jako je např. doba vykonání dílčí části programu. Kritika této práce dle autora spočívá v kritice použití návrhových vzorů (viz kap. 6.4). Části a závěry této práce jsou využity v kapitolách 4 a 6.

Disertační práce [151] obhájená na Univerzitě obrany v Brně v roce 2007 pojednává o zvýšení spolehlivosti RTOS pomocí průběžné diagnostiky detekce a lokalizace poruch. Autorem navrhnutá metoda se zakládá na implementaci vlákna typu watchdog k vláknům zajišťujícím dílčí funkcionalitu (z mého pohledu se jedná o implementaci návrhového vzoru). V uvedené implementaci návrhu s následným testováním však shledávám hrubé zanedbání možnosti vykonání jiného pořadí vláken, než autor uvádí, což má za následek vyhodnocení chyby. Autor implementuje program pomocí MDA metody, kdy z *platformově nezávislého modelu* (v mém pojetí jde o návrhový vzor) postupně generuje kostru programu v daném programovacím jazyce. Poukazuje také na množství problémů této metody, tedy možnost skrytých chyb, které tímto procesem mohou vzniknout. Některé myšlenky, tvrzení a poznatky jsou využity v kapitolách 1, 4, 5 a 8.

Stavím také na výzkumu Dr Ammon H. Eden, který se se svým týmem zabývá softwarovým inženýrstvím a přispěl také v oblasti návrhových vzorů a modelování programu. Za přínosný pokládám článek [92], kde je formálně dokázán přínos návrhových vzorů z hlediska implementace změn v programu. V [44] popisuje prototyp

nástroje pro automatizovanou implementaci návrhových vzorů a v [99] popisuje automatizovanou verifikaci přítomnosti návrhového vzoru v kódu, což je využitelné v procesech verifikace kódu programu a jeho vlastností. Moje práce se zakládá na výše uvedených poznatcích, protože bez nich by bylo obtížné implementovat návrhové vzory a jejich aplikace dle normy IEC 61508.

3.2 Inovátorská řešení

V této kapitole uvedu stručně odkazy na relevantní výzkum a projekty zabývající se stejným či podobným tématem. Tyto práce přistupují k danému problému z jiného nebo podobného úhlu, doplňují danou problematiku, a poskytují tak další možnosti zlepšení spolehlivosti a funkční bezpečnosti RTOS systému.

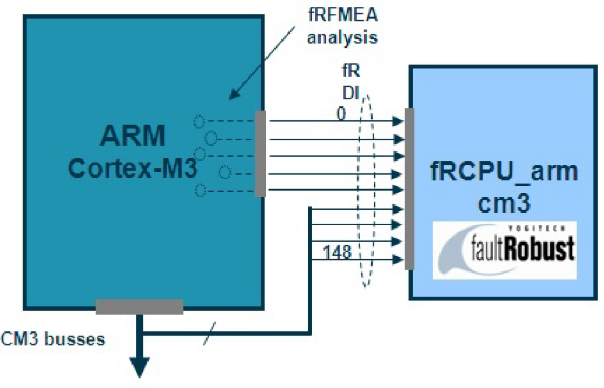
3.2.1 Yogitech's faultRobust-CPU

Pro procesory ARM Cortex M3 vyvinula firma Yogitech rozhraní pro připojení fRCPU. fRCPU je periferie procesoru, která monitoruje CPU stylem *white-box* a zároveň je napojen na speciální monitorovací port procesoru fRDI obsahující cca 150 signálů (viz obr. 3.1). CPU je rozděleno na zóny, kterým je přidělen stupeň rizika chyby, z čehož je pak statisticky vypočítána pravděpodobnost chyby. fRCPU monitoruje vnitřní části CPU, jako jsou registry, stupně pipeline a různé debugovací informace. Jsou zde zachyceny i chyby, které by pomocí např. lockstep architektury nebyly zaznamenány. fRCPU dosahuje SIL 3 a dle provedených testů SFF dosahuje $99,2\ \%$.

fRCPU je implementováno pomocí 150 nm technologie a plocha celé periferie je spočítána na $0,32\ mm^2$. To je několikrát méně než plocha dalšího CPU $1,6\ mm^2$, což se hlavně projeví ve vícejádrových architekturách. Více je uvedeno v [85]. V roce 2016 avšak Intel koupil firmu Yogitech, a tento projekt tedy už dále nejspíše nepokračuje.

3.2.2 Mikrokontroléry Hercules

Firma Texas Instruments nově nabízí řadu mikrokontrolérů pod označením Hercules. Tyto mikrokontroléry jsou určeny pro aplikace do oblastí safety-critical, medical, industrial a transportation. Jsou postaveny na jádře ARM Cortex-R, což je jádro navržené pro bezpečné a spolehlivé systémy [49]. Hlavní použité fault-tolerant metody jsou CPU lockstep, hardwarový diagnostický subsystém včetně monitorování hodinového signálu, hardwarová LBIST diagnostika prováděná při startu i částečně při běhu CPU, zabudovaný ECC kontrolér přímo do CPU pro zachycení náhodných chyb při přenosu dat po datové sběrnici procesoru a hardwarové subsystémy pro testování



Obr. 3.1: Připojení fRCPU na CPU [85]

periferií. Více je uvedeno v [57]. Jedná se tedy o mikrokontroléry se zabudovanými technikami funkční bezpečnosti na hardwarové úrovni, které umí detekovat přechodné chyby, avšak neumí detekovat chyby designu a chyby RTOS.

3.2.3 Hardwarový plánovač pro detekci chyb

Hardware-Scheduler (Hw-S) je modul implementovaný v hradlovém poli, který detekuje určité chyby plánovače. Jako chyba se počítá přepnutí vlákna mimo časové okno vymezené systémovým časovačem a povolenou dobou přepnutí vláken, přepnutí na nesprávné vlákno a překročení samotné doby přepnutí vlákna. Modul je přímo napojen na adresovou sběrnici procesoru, na signál ze systémového časovače a adresovou sběrnici, ze které rekonstruuje aktuálně vykonávané vlákno. Autor vyzkoušel modul v kombinaci s Plasma procesorem implementovaným v hradlovém poli a chybu simuloval zvětšováním amplitudy poklesu napájení na frekvenci 0,3 MHz. Více je uvedeno v [133].

Implementovaný modul dokáže tedy detekovat chyby plánovače, které jsou způsobeny přechodnými chybami v podobě propadů napájecího napětí, časem řízeného RTOS monitorováním adresové sběrnice procesoru a signálu ze systémového časovače. Modul tedy bude při použití v událostmi řízeném systému nebo mixovaném systému generovat mnoho planých poplachů. Pořadí vykonávání vláken je nutné předem do modulu napevno nastavit a dynamicky vzniklá vlákna budou vyhodnocena jako planý poplach. Více informací nebo pokračování k tomuto přístupu jsem nenašel.

3.2.4 Hardwarový multi-watchdog

Tento výzkum se zabývá použitím mnoha watchdog jednotek implementovaných v hradlovém poli. Každé vlákno je tedy hlídáno svou vlastní watchdog jednotkou

skládající se z watchdog časovače a prioritního dekodéru/enkodéru napojeného na adresovou sběrnici CPU. Výhody a nevýhody použití watchdogu jsou probrány v kapitole 2.2.3. Více je uvedeno v [104].

3.2.5 Runtime měření času vykonání vláken

Měření času vlákna v běžícím stavu hlídá správnou funkci vykonávání vlákna, aby jeho běh nepřekročil jeho deadline. Měření času vykonání vláken v reálném čase je náročné na externí výpočetní zdroje, obzvláště v systému s více vlákny. Dalším přístupem je problém obrátit a počítat čas vlákna v blokovaném stavu. Tento přístup je popsán v [94].

3.2.6 Diagnostický softwarový systém

V [151] se autor zabývá detekcí chyb v Linux OS s RTAI patchem pro zlepšení real-time vlastností. Implementuje monitor parametrů real-time systému a watchdog do jádra systému. Detekuje tím chyby, které způsobí změny v registrech a adresách. Monitor je ale omezen pouze na Linux OS s RTAI patchem a snižuje výpočetní výkon systému.

3.2.7 Monitor řešitelnosti plánovače

V [95] jsou pro detekci chyb využity výpočty řešitelnosti plánování (angl *feasibility*) v závislosti na běžících úlohách a jejich nejhorších časech. Tato softwarová kontrola se provádí v rámci úkonu přepnutí kontextu. Metoda odhalí chyby, které způsobí zmeškání deadline a jiné chyby meziprocesové komunikace, avšak spočítat funkci užitečnosti (angl. *utility function*) je pro některé algoritmy (např. RMS) obtížné, ne-li pouze orientační.

3.2.8 Řízení dle scénáře

[24] představuje zajímavou metodu kontroly řízení *safety-critical* systému, která spočívá ve vytvoření modelu aplikace pomocí časové Petriho sítě obsahující zpoždění. Hodnoty těchto zpoždění a správnost celého modelu se spočítá offline, přičemž autoři tvrdí, že zamezení chyb vyplývajících z anomálií RTOS může také spočívat právě ve vložení těchto definovaných zpoždění, aby byl čas výkonu funkcí definovaný. Tato zpoždění jsou pak za běhu injektována a upravována pomocí kontrolního subsystému. Tato metoda byla testována na Trampoline OS, který obsahuje fixní plánovač.

3.2.9 Softwarová řešení

V software lze detekovat některé případy chyb RTOS (*deadlock*) pomocí knihoven zapouzdřujících práci s vlákny a synchronizačními mechanismy. Tato řešení se nasazují spíše na platformách s vyšším výpočetním výkonem a komplexnějším prostředím (např. Linux). Tyto knihovny potom dokonce detekují možný vznik těchto chyb, které následně nepovolí zamezením vykonání daného vlákna. Princip spočívá v implementaci obálek vláken, na jejichž základě si dokáží tyto algoritmy (např. Tarjanův algoritmus) vytvořit aktuální mapu závislostí vláken a otestovat, zda je požadovaná operace RTOS validní.

3.2.10 Implementace specifikace v hardware

Verifikaci invariant nebo jiných specifikací subsystému vyjádřených např. v *past-time Metric Temporal Logic* je možné implementovat v hardware (např. FPGA). Tyto specifikace potom v reálném čase vyhodnocují svoji logickou hodnotu a signalizují rozpor. [136] ukazuje, jak lze tyto specifikace implementovat a redukovat jejich náročnost na zdroje.

3.2.11 Hardware kontrolní subsystém

[6] uvádí metodu kontroly vykonávání funkcí pomocí hardwarového subsystému (angl. *hardware-assisted run-time monitoring*). Na příkladu funkce pro šifrování předvádí, jak napojit vytvořený *soft-core* v hradlovém poli na subsystém monitoru pomocí odboček z instrukčního procesu (angl. *pipe-line*) pro získání aktuální vykonávané adresy. Jako model použili konečný stavový automat (FSM) s převodními tabulkami počátečních a koncových adres. Autoři se tímto záměrem zaměřili na detekci bezpečnostních útoků na tyto algoritmy.

Tento přístup je nejvíce podobný mému záměru. Plánuji ale zakomponovat navíc atribut času do modelu a formalizovat modelovací nástroj. Důležitým poznatkem je také možnost využití této techniky pro detekci kyber-útoků.

3.2.12 RTOS-G

[119] prezentuje RTOS-Guardian jako IP subsystém, který detekuje chyby preemptivního round-robin plánovače Plasma-RTOS. Oproti Plasma *soft-core* zabírá kolem 10 % místa. Monitor získává data z adresové sběrnice procesoru, musí tedy odvodit nově přepnutou úlohu z důvodu synchronizačních mechanismů pouze z adres volaných funkcí. Funguje na principu paralelního plánovače, jehož výstup se porovnává s monitorovanou rekonstruovanou skutečností. Systém byl testován podle mezinárodního

standardu IEC 61.000-4-29 (definuje pravidelné poklesy, krátké výpadky a přechodná přepětí v napájení). Výsledky ukazují zlepšení diagnostického pokrytí z původních 41 % na 98.7 % se současným snížením doby detekce až na 2 % oproti původní době detekce samotného RTOS.

Tento přístup, i když je jeho popis vágní, ukazuje podobný pohled na problematiku jako moje práce. Zajímavé jsou ale výsledky testování dle uvedeného standardu vedoucí ke zlepšení diagnostického pokrytí, což implikuje přínos tohoto přístupu. Bohužel jsem nenašel další informace o pokračování tohoto projektu. Navíc zdrojové kódy a vnitřní uspořádání také nejsou dostupné.

4 FORMÁLNÍ METODY

Jedno z doporučení a požadavků na návrh zařízení certifikovaných dle standardu IEC 61508 je použití formálních a semiformálních modelů. Každý vytvořený model vytváří určitý pohled na daný systém, přičemž potlačuje nesledované vlastnosti. Model tedy v určité míře aproximuje skutečnosti, resp. struktury a funkce zařízení. Účel vytvoření modelu, tedy jeho funkce, může být dokumentační, návrhový, simulační a verifikační.

Tato kapitola slouží jako rešerše formálních metod pro modelování a popis *safety-critical* systémů, a to jak pro akademické, tak pro komerční využití. Jelikož existuje těchto metod mnoho, vyberu pak pro mou práci tu nejvhodnější z hlediska věrohodného popisu aplikace, implementovatelnosti pomocí syntetizovatelných jazyků, možnosti nasazení verifikačních metod (*model-checking* nebo *theorem proving*) a použitého matematického základu.

Verifikační modely popisují systém pomocí formálních modelů. Na takovém modelu je možné testovat splnění definovaných vlastností. Verifikace používá matematické modely a analýzy, takže je možné deklarovat platnost pro daný definiční obor dat. Tato analýza se používá hlavně k verifikaci softwarové části systému. *Safety-critical* aplikace jsou většinou implementovány pomocí programovacích jazyků na základě imperativního paradigmatu. Proto je nutné obezřetně formulovat formální modely, které mají základ ve funkcionálním paradigmatu nebo mají odlišnou sémantiku.

Právě dle úrovně integrovatelnosti formálních metod do procesu návrhu aplikace se dělí na [140]:

- úroveň 0 – formální metody jsou použity pouze pro popis systému,
- úroveň 1 – pokud jsou navíc použity i pro verifikace funkce/í,
- úroveň 2 – pokud jsou systém a jeho funkce modelovány a verifikovány pomocí těchto metod.

Vědecké modelování je obecná metoda sloužící k získání popisu daného systému a jeho částí na zvolené úrovni abstrakce. V oblasti embedded systémů s RTOS je mnoho možností uplatnění této metody, a to např. vytvoření modelu programu za účelem simulace funkčnosti, verifikace bezchybného provozu, implementace, nebo dokonce získání modelu vlastního RTOS. Výrobci vývojových prostředí (dnes už tzv. *softwarových továren* [58]) stále více začleňují abstraktní notace (např. UML nebo AADL) pro dokumentování kódu programu, ale i pro jeho implementaci. Výzvou pro tato prostředí zůstává hlavně implementace bezchybného bezpečného programu, formální specifikace požadavků na program [8] a také integrace nově vznikajícího trendu vývoje software dle specifikace, resp. *modelem řízená architektura* [59].

Při tvorbě rešerše vycházím z knih a akademických příspěvků, jako je kniha [9].

Dále čerpám informace z programátorských fór, volných internetových článků a prezentací dostupných nástrojů, jakožto pohled z praxe. Také vycházím z výukových materiálů univerzit, jako je např. Centrum návrhu elektronických systémů v Berkeley [46], Institut výzkumu počítačového inženýrství v Rennes [132] či Oddělení výpočetního software na McMaster univerzitě [72].

Problematika modelování programu a jeho formální specifikace je stále aktivní, a to hlavně díky nejednotnosti standardu používaných modelovacích nástrojů, což pramení z omezení jednotlivých nástrojů a vyšších požadavků na univerzálnost z hlediska jejich funkcí. V následujících podkapitolách uvádím stručnou rešerši dostupných nástrojů s ohledem na použití v oblasti vestavných systémů pracujících v reálném čase.

4.1 Temporální logika

Temporální logika a její odvozeniny patří k nástrojům pro formální zápis požadované specifikace na systém, tedy jeho vlastnosti a funkce. Potom je možné použít jinak vágně nadefinované specifikace pro verifikaci RTOS systému (viz např. [47]). Temporální logika se vyvinula z modální logiky a přídává operátory eventuálně (angl. *eventually*) a vždy (angl. *always*). Dále definuje operátory další (angl. *next*) a dokud (angl. *until*). Konceptem lineární temporální logiky (LTL) je, že každá instance má pouze jedinou budoucnost. Naopak u *computation tree logic* (CTL), která vychází z *branching-time temporal logic*, může nastat z jedné instance různá budoucnost, což spíše předurčuje tento nástroj pro popis ne-deterministických systémů [128].

LTL tedy může sloužit pro vytvoření rovnic definujících atributy dosažitelnost (angl. *reachability*), bezpečnost (angl. *safety*) a provozuschopnost (angl. *liveness*). Vědeckou aktivitu v této oblasti dokládá množství vylepšení teorie temporální logiky. [69] například zavádí *signal temporal logic*, která umožňuje popsat signál v čase pro verifikaci systémů v FPGA. *Metric temporal logic* dále obohacuje sadu operátorů LTL a zavádí operátory pro práci s událostmi z minulosti a budoucnosti. MTL omezuje události v čase a za určitých omezení je vhodná ke specifikaci funkcí a vlastností kladených na *real-time* systémy [87].

4.2 UML

Objektová notace UML patřící mezi semi-formální metody je velmi rozšířená právě díky podpoře vývojových nástrojů a také díky své srozumitelnosti. Umožňuje modelovat systémy a procesy. V programátorské praxi se tento nástroj nejčastěji používá

pro účely dokumentace a v nových moderních nástrojích už i jako model pro implementaci struktury programu (např. Eclipse s příslušnými pluginy). Tento nástroj neobsahuje některé výrazy pro popis programu, avšak novější verze, např. 2.5 z roku 2015, už definuje vztahy, jako je např. dědičnost nebo přetěžování [60]. Také akademici tento nástroj stále vylepšují za účelem vyšší komplexity a přesnosti pro popis skutečnosti.

4.3 AADL

AADL patří k semi-formálním metodám popisu architektury a návrhu *safety-critical* aplikací obzvláště v oblasti letectví [67]. Obsahuje knihovny, resp. axiomy pro popis architektury, definici propojení subsystémů, ale také nástroje pro popis potenciálních chyb v dané části subsystému [36]. Na základě tohoto popisu lze pak dle zadaných hodnot pravděpodobnosti vyčíslit teoretickou spolehlivost systému (kalkulace počítá s modelem Markovova řetězce), a tedy dokonce nahradit chybové diagramy (angl. *fault-tree diagram*) [35]. Definováním parametrů chování subsystémů pak také lze provádět statickou analýzu, např. analýzu plánování (angl. *schedulabilty test*) specifikovaných úloh (viz [64]) nebo použít jiné nástroje formální verifikace.

Tento nástroj se neustále vyvíjí jak v komerční (organizace SAE International vydala v roce 2012 modifikovanou verzi z původní verze z roku 2004), tak převážně v akademické sféře (hlavně na univerzitě Carneige Mellon). Vznikají tedy další nástroje rozšiřující tento ekosystém. [38] navrhnul metodu pro testování modelu aplikace pomocí Markovova řetězce. [19] uvedl metodu testování dosažitelnosti modelu v čase pomocí metody Monte Carlo. Pro simulaci a generování kódu z modelu slouží vyvíjený nástroj OSATE plugin do Eclipse [48], plugin Ocarrina nebo simulační nástroj *Cheddar*. Pomocí AADL je také možné modelovat design pro FPGA čipy nebo SoC obvody, a dokonce jej implementovat na úrovni propojených funkčních bloků. Koncepce AADL a designu FPGA je natolik podobná, že nevznikají závažné problémy v propojení těchto prostředí na syntaktické úrovni a nevznikají tedy kolize jednotlivých paradigmat [15].

Tato metoda by tedy měla sloužit jak ve fázi návrhu, tak i ve fázi specifikace a implementace. Je tedy v souladu s metodikou návrhu dle modelu (angl. *Model-Based Design*) a zapadá i do kontextu návrhu modelu architektury (angl. *Model-Driven Architecture*) [80]. Tímto se minimalizují chyby lidského faktoru pramenící z kopírování heterogenních dat.

4.4 Matlab Simulink

Nástroj Matlab, resp. Simulink patří do semi-formálních metod popisu a simulace systémů. Jelikož se ale v praxi stále častěji nasazuje, považuje se za *průmyslový standard*. Tento nástroj je aktivně rozšiřován a zlepšován firmou MathWorks, a proto obsahuje různé pluginy, např. Embedded Coder pro generování kódu z modelu, PolySpace Prover pro formální verifikaci modelu a PolySpace Bug Finder pro hledání defektů, resp. potenciálních chyb přímo v kódu vygenerované aplikace v jazyce C nebo C++.

Struktura generovaného kódu připomíná standardní strukturu S-funkce v prostředí Simulink. Obsahuje tedy funkci *init* pro inicializaci proměnných a funkci *step* pro spočítání výstupních hodnot jednotlivých bloků, resp. propagaci signálů v jednom diskrétním kroku. Tato architektura je postavena na FreeRTOS operačním systému, který se generuje spolu s kódem aplikace. Použité funkce a datové typy odpovídají zamýšlené univerzálnosti použití, takže výsledný kód je větší a má delší čas vykonání než ručně psaný ekvivalent [111].

Příkladem použití této metodologie může být např. [75], kde autoři pomocí Matlab toolboxů Simulink a Stateflow namodelovali, otestovali, simulovali (tzv. *hardware-in-the-loop* simulace) a vygenerovali finální kód pro procesor pomocí toolboxu Embedded Coder. V nastavení je také možné omezit generátor na sadu příkazů v souladu s MISRA C, a dokonce nahradit některé funkce za uživatelem zvolené ekvivalenty. Tyto funkce zvyšují spolehlivost výsledného systému a umožňují průchod certifikačním procesem.

4.5 Přechodové systémy

Přechodový systém (angl. *transition system*) je obecně abstraktní matematický stroj vycházející z teorie orientovaných grafů. Pro praktické použití se používají jeho odvozeniny, např. konečný stavový automat (angl. *finite state machine*), který už má definovanou sadu stavů a přechodů, a tedy simuluje jednoduchý počítač. V praxi se tento nástroj používá k formalizaci řídicí rutiny. V akademickém prostředí se používá také pro simulaci chování a verifikaci real-time systémů [4], resp. jeho forma využívající globální proměnnou pro simulaci času [125].

Nejvhodnější variací přechodového systému, resp. konečného automatu, je deterministický konečný automat používaný hlavně pro verifikaci procesu (např. postup signálu nebo algoritmu). Je definován jako uspořádaná pětice $(Q, \Sigma, \sigma, q_0, F)$, kde

- Q je konečná neprázdná množina stavů,
- Σ je konečná neprázdná množina vstupních symbolů,
- $\sigma : S \times \Sigma \rightarrow S$ je přechodová funkce,

- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina přijímacích stavů.

Pro analýzu programu pomocí přechodových systémů z hlediska obsahu chyb RTOS, jako je *deadlock*, *livelock*, *starvation* a *race condition* je možné požadavky definované v temporální logice pro *safety* („nic špatného se nestane“) a *liveness* („správný stav se nakonec uskuteční“) testovat pomocí formální metod [71]. Autor také uvádí možnost dekompozice konkurentního systému za účelem tvorby značeného přechodového systému (angl. *labelled transition system*).

[82] uvádí návrh časového přechodového systému (angl. *clock transition system*), který aspiruje na modelování aplikací jako časových systémů pro modelem řízený návrh. Autor také prezentuje, že časová Petriho síť nebo časový automat (angl. *timed automaton*) mohou být převedeny do tohoto modelu. Bez odvození navrhuje, že pro analýzu tohoto modelu mohou být použity nástroje obdobné jako pro standardní přechodné systémy. V závěru ovšem zmiňuje omezení pro konkurentní systémy.

4.6 VHDL

Jazyky VHDL a Verilog se používají pro definici architektury a chování aplikace pro hradlová pole (FPGA). Ač se jedná o notaci, tedy o semi-formální metody, je jazyk VHDL považován mnohdy za formální, a to díky jeho komplexní definici. Jazyk VHDL se oproti Verilogu používá hlavně na akademické půdě právě z důvodu jeho jednoznačnosti. Definici architektury cílové aplikace lze převést (např. do SystemVerilog nebo Symbolic Model Verification) na formální model [89] vhodný pro následnou verifikaci. Modelování a simulace programu pro procesor jsou také umožněny pomocí různých nástrojů, které podporují spojení kódu s hardware simulace na definované úrovni (např. na funkčním modelu procesoru, soft-core nebo funkčním modelu RTOS) [121].

Programovací prostředí VHDL tvoří pouze část možností celého ekosystému jazyka. Existuje také např. rozšiřující knihovna VHDL-AMS (IEEE 1076.1999), která zavádí matematické operace pro modelování a simulaci diskrétních systémů [31]. Se současným tlakem na rozvoj tohoto modelovacího nástroje v důsledku existence mnoha IP komponent vznikají nástroje, které umí konvertovat kód z jiných jazyků (např. HDL) právě do VHDL-AMS, což se pomalu stává budoucím standardem [61].

4.7 Petriho sítě

Petriho sítě v různé variaci se používají jako formální modelovací nástroj konkurentních systémů převážně v akademické sféře. Teorie vychází z modelu orientovaného

grafu, na kterém je pomocí operační sémantiky definována operace se značením (angl. *token game*). Tento nástroj je dlouhodobě zkoumán a zdokonalován, příkladem může být definice stochastického rozšíření Petriho sítě [130] a ukázka modelování s využitím tohoto rozšíření pomocí skriptovacího jazyka [100]. Základní Petriho síť, ze které vychází všechna rozšíření, je definována jako pětice (S, T, F, M_0, W), kde

- S je konečná množina míst,
- T je konečná množina přechodů,
- F je konečná množina hran $F \subseteq (S \times T) \cup (T \times S)$,
- $M_0 : S \to \mathbb{N}$ je počáteční značkování,
- $W : F \to \mathbb{N}^+$ je množina vážených hran.

Petriho síť může být kromě modelování a simulace použita také k statické analýze a verifikaci (např. analýza plánovacího procesu RTOS [1]). Také už jsou aktivně zkoumány možnosti modelování, simulace a analýzy běhu úloh na více-procesorových architekturách (viz [56]). Poznatky plynoucí ze základního výzkumu Petriho sítí se snaží reflektovat všechny aspekty pro aplikaci této teorie, avšak pro pochopení celé problematiky je potřeba definovat některé atributy a operace i pomocí jiné teorie, např. přechodových systémů, se kterými jsou Petriho sítě spjaty. Potom lze vyšetřit vlastnosti sítě, jako je omezenost (angl. *boundedness*), provozuschopnost (angl. *liveness*), bezpečnost (angl. *safety*), ustálené stavy (angl. *invariants*), dosažitelnost (angl. *reachability*) aj. [96]. Splněním definovaných vlastností lze potom model aplikace prohlásit za formálně verifikovaný.

Z hlediska použití v oblasti *real-time* systémů mohou být Petriho sítě nasazeny také za účelem verifikace (viz [83]), statické analýzy dosažitelnosti (viz [139]), simulace (viz [34]) nebo pouze formalizace (viz [52]). I když výpočetní nároky kvůli stavové explozi mohou být pro některé operace velmi náročné (viz [14]), snaží se akademici najít vhodný způsob pro detekci chyb RTOS, jako je deadlock, livelock a porušení časového determinismu plánovače (viz [143]). Některé přístupy dokonce kombinují různé formální metody za účelem redukce stavové exploze při testování modelu (viz [32]). Dalším použitím může být přímé řízení procesu v reálném čase (viz [34]), avšak samozřejmě po úpravě teorie.

Základní Petriho síť může být obohacena o atribut času, jedná se tedy o časovou Petriho síť (viz [112]). Pro nasazení verifikačních metod na tuto síť navrhnul [112] postup převodu na graf tříd stavů (angl. *state class graph*) a až následné vyšetření vzniknuvších skupin. Barevná Petriho síť umožňuje definovat atributy značení a hraniční podmínky, resp. poskytuje více možností modelování (viz [146]). Pomocí barevné sítě je definována právě časová Petriho síť typu časové značky markeru. Druhý typ časové Petriho sítě definuje globální hodiny (více v [102]). Jednoznačné řešení dosažitelnosti a omezenosti však u odvozenin z časové Petriho sítě nelze nalézt (viz [112]).

Pro implementace Petriho sítě jako abstraktního nástroje je možné využít konstrukcí programovacích jazyků nebo ji implementovat do VHDL. Naivní možností implementace do VHDL je využívat pro každý objekt typu *place* pouze jedno *flip-flop* hradlo a pro přechody pouze kombinatorní logiku. Druhou možností je využívat paměťové hradlo pro objekt typu *place* i pro *transition*, jak uvádí [122]. Implementaci Petriho sítě je možné použít pro simulaci (viz [123]), ale i k řízení daného procesu (jak prezentuje v [54]). VHDL kód pro implementaci je možné také generovat pomocí nástrojů softwarového inženýrství z vytvořeného modelu.

Při statické analýze dosažitelnosti systému umí nástroje pro základní Petriho sítě detekovat *deadlock* jakožto vyšetření *T-invariant* (viz [97]) a vlastnosti *liveness*. [3] navrhuje pro FMS systémy kontrolovat speciální struktury (angl. *siphon*) v Petriho síti pro detekci potenciálního *deadlocku*. Standardní techniky analýzy dosažitelnosti nedokáží u časových Petriho sítí verifikovat atribut času, proto [139] navrhuje použít místo klasického stromu tříd stavů (angl. *state class graph*) koncept tříd stavů s časovou značkou (angl. *clock-stamped state class*).

4.8 Zhodnocení

Literární průzkum v oblasti formálních metod používaných při návrhu a vývoji software přinesl náhled do rozsáhlé oblasti, ve které vznikají stále nové přístupy a techniky pro tvorbu modelu programového vybavení za účelem verifikace, dokumentace a implementace vlastního kódu. Literaturu lze rozdělit na teoretický výzkum jednotlivých formálních nástrojů a na aplikaci této teorie pro reálné systémy. Jelikož je použití formálních a semi-formálních metod vyžadováno či doporučeno standardem funkční bezpečnosti, reaguje výzkum v této oblasti na vzniklé požadavky plynoucí z aplikací. Kromě aplikačních výhod má integrace formálních metod do vývoje systému výhody společenské ve smyslu ustanovení jednotného formálního popisu vyvíjeného systému, kterému rozumí všichni účastníci a jehož dílčí části mohou být dále znovu použity v podobných aplikacích.

5 HARDWAROVÁ IMPLEMENTACE ALGORITMŮ

V této kapitole rozeberu přístupy k testování vestavných systémů z hlediska napojení testovacích subsystémů. Zároveň zde nastíním možnosti implementace logiky do hardware za účelem zvýšení spolehlivosti těchto subsystémů (diversifikace), a tím i celého systému. Tyto dvě oblasti spolu souvisí, jelikož implementace testovacího subsystému závisí na rozhraní mezi tímto subsystémem a vlastním výkonným subsystémem. Proto ještě provedu krátkou rešerši tzv. *soft-core*, tedy procesorů implementovaných v hradlovém poli.

5.1 Rozhraní testovacího systému

Současné vestavné systémy používají různé přístupy pro testování, monitorování a diagnostikování svého chování. Cílem je získat ucelené informace o stavu systému při co nejmenším zásahu do jeho chodu. V případě systému reálného času je potřeba testovat determinismus monitorovacího procesu, přičemž je možné si dovolit využít více výpočetního výkonu. Příkladem takovéto monitorovací techniky je zabudovaný test (angl. *built-in self test*), který se spouští po startu zařízení nebo v pravidelných intervalech. Tento typ testování zvyšuje aspekt diagnostického pokrytí (angl. *diagnostic coverage*).

Při požadavku na co nejmenší invazivnost monitorovacího procesu se používají tzv. *in situ* monitorovací techniky. Jedná se o techniky monitorující systém na hardwarové úrovni, např. čtením procesorových registrů, nebo techniky založené na použití výpočetně krátkých instrukcí, např. zápis hodnoty do periferní jednotky. Výhoda těchto technik tedy potom spočívá v menší zátěži monitorovaného subsystému plynoucí z monitorovacího procesu a také menší závislosti na subsystému monitorovacím.

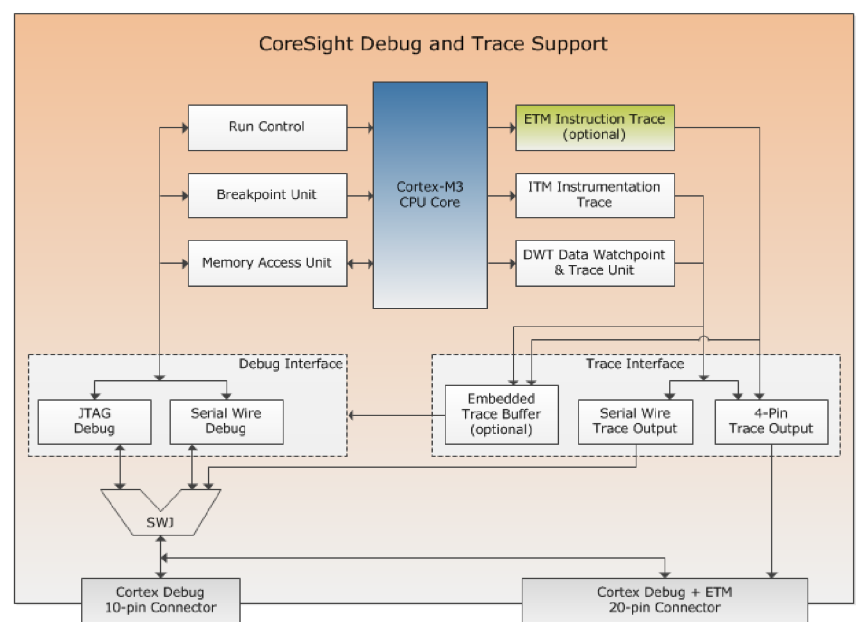
V dalších podkapitolách popíši přístupy pro monitorování chodu procesoru, které lze klasifikovat jako neinvazivní, a dále uvedu i techniky invazivní.

5.1.1 ARM Cortex CoreSight a JTAG

CoreSight je subsystém (viz obr. 5.1), kterým je vybavena většina mikrokontrolérů postavených na jádře *ARM Cortex*. Jedná se o IP core obsahující subsystém poskytující informace o vnitřním stavu procesoru a umožňující i trasování instrukcí pomocí omezené paměti. Technologie JTAG popsaná v IEEE 1149.1 (IEEE Standard Test Access Port and Boundary Scan Architecture) slouží pro připojení se k debugovací

jednotce. Jedná se tedy o průmyslový standard diagnostického rozhraní integrovaných obvodů s vysokým stupněm integrace (FPGA, CPLD, MCU, CPU atd.).

Tato metoda monitorování procesorového systému patří k nejméně invazivním, avšak vyžaduje velký výpočetní výkon (v reálném čase) připojeného systému s velkou kapacitou paměti. Metoda je výhodná také z toho důvodu, že dokáže diagnostikovat vnitřní stav procesoru i v případě zastavení vykonávání programu. Na druhé straně se tato technologie používá hlavně ve fázi ladění a v produkční fázi se už nevyskytuje, protože umožňuje také velmi jednoduše procesor ovládat, tedy např. pozastavit jeho činnost. Pokud je možné zajistit vysokou spolehlivost obslužného monitorovacího subsystému, jedná se o nejvhodnější a nejméně invazivní možnost monitorování systému, která je vhodná k dalšímu výzkumu možností implementace.



Obr. 5.1: Schéma technologie *ARM CoreSight* [73]

5.1.2 Soft-core

Další z nejméně invazivních metod je implementace procesoru v hradlovém poli, kde má monitorovací subsystém možnost napojit se přímo na registry tohoto procesoru a na požadované periferní jednotky. Tato koncepce také umožňuje celý systém simulovat pomocí nástrojů pro vývoj architektur pro hradlová pole. K tomu je ale potřeba mít zdrojové kódy celého soft-core v některém popisném jazyce (např. VHDL), což je v rozporu se zájmy jejich výrobců, kteří uvolňují soft-core jako IP komponenty. Jediný open-source soft-core, který jsem našel, se nazývá Plasma a v základních variantách i Leon (více o této problematice v kap. 5.3).

5.1.3 Koprocesor

V multiprocesorových systémech je možné jedno z jader vyčlenit a použít jako koprocesor, tedy vyčlenit kontrolní algoritmy mimo hlavní jádro. Výhodou v tomto případě je běh monitorovacího jádra nezávisle na běhu hlavního jádra a sdílená paměť až na úroveň cache, kterou je ale nutné naparametrovat v obou jádrech. Pro detekci zaseknutí běhu hlavního jádra je nutné implementovat do monitorovacího jádra nezávislý operační systém. Obě jádra jsou provozována ze stejného napájení a hodinového signálu, takže trpí na poruchy se společnou příčinou (angl. *common cause failure*. Nevýhodou také je, že monitorovací jádro musí stihnout upočítat algoritmy v době mezi událostmi z hlavního jádra a synchronizovat komunikaci přes sdílenou paměť.

5.1.4 Standardní komunikační sběrnice mikroprocesoru

Mezi méně invazivní techniky patří metoda získávání informací ze systému založená na zapisování požadovaných hodnot (informací) do speciální periferní jednotky. Takováto periferní jednotka je integrována do procesorového systému pomocí procesorové sběrnice, která tuto jednotku řídí, adresuje a poskytuje jí data.

V software se tato technika implementuje pouze jako zápis na určité místo paměťového prostoru, a to v minimální variantě jednou instrukcí, což zajišťuje minimální zátěž na monitorovaný systém a determinismus monitorovacího procesu. V hardware je monitorovací subsystém implementován jako mapovaná periferie procesoru. V závislosti na integraci, tedy použité sběrnice, jsou dány vlastnosti zápisu, jako je rychlost a funkční bezpečnost. Mezi takové sběrnice patří např. APB, AHB a AXI pro platformu ARM; PCI pro platformu x86; Avalon pro platformu Nios.

Modifikací této metody je použití komunikační sběrnice mikrokontroléru (např. UART, SPI, nebo dokonce Ethernet). Tato modifikace sice používá standardní sběrnice, které jsou lehce dostupné, avšak zvyšuje výpočetní náročnost procesu monitorování o obsluhu periferních jednotek zajišťujících danou komunikaci.

5.1.5 Software tracing

Mnoho RTOS poskytuje mezi svými knihovnami také různé diagnostické nástroje (např. FreeRTOS poskytuje FreeRTOS Plus Trace). Tyto nástroje poskytují informace o systému z hlediska operačního systému, tedy přepnutí vlákna nebo použití synchronizačního objektu. Tyto informace pak mohou být exportovány a zpracovány. Nevýhodou je vyšší výpočetní zátěž na monitorovaný systém a nutnost jeho součinnosti. Z této podmínky vyplývají také nedostatky této metody, a to nemožnost monitorování při přerušení vykonávání (např. *deadlock*).

5.2 Hardwarová implementace algoritmů

Všechny algoritmy, které jsou nezbytné pro chod procesoru a jeho periférií, jsou implementovány přímo do křemíkové báze (např. řadič procesoru, jednotka správy paměti aj.). Jedná se tedy o tzv. ASIC obvody. Další možností je využít hradlová pole (trvalá nebo programovatelná), která umožňují změnu struktury hardwarového propojení logických buněk (angl. *logic cell*) na základě upravovatelného předpisu (angl. *bitstream*). Tímto se jeví jako ideální volba pro hardwarovou implementaci algoritmů v prostředí výzkumu.

Hustota transistorů stoupá s pokročilostí výrobní technologie, která v současné době dosahuje mezí 14 *nm*, resp. 10 *nm* u ASIC obvodů a 16 *nm* u FPGA. Důvodem stále častějšího nasazování hradlových polí spočívá ve změně paradigmatu, a to ze software *pipe-line* na paralelní běh implementovaných úloh. Navíc použití FPGA pro algoritmy vedle klasické ASIC implementace, tzv. systém na čipu (SoC), zvyšuje funkční bezpečnost díky diversifikaci použitých technologií. Také existuje mnoho akademických (např. [110]) i komerčních (např. [28]) řešení snižujících riziko defektu hradlových polí jako komponentů.

Obě technologie lze s výhodou kombinovat, a to v podobě tzv. SoC čipů, které většinou sdružují dvě ASIC procesorová jádra (tzv. HPS) a hradlové pole. Mezi těmito subsystémy lze přenášet data pomocí HPS-to-FPGA rozhraní. Na druhé straně toto rozhraní neumožňuje přímo vyčítat vnitřní registry procesory. Tuto nevýhodu by vyřešila implementace celého procesoru do FPGA části (viz kap. 5.3).

5.2.1 Možnosti prostředí FPGA

Prostředí hradlových polí nabízí paralelní zpracování signálů/algoritmů, přičemž má blízko k ASIC prostředí. Používají se zde však jiná programovací prostředí. VHDL je prostředí, které svým vyčerpávajícím popisem zaručuje očekávané chování při správné implementaci. Jelikož se jedná o jazyk využívaný i pro modelování systémů (vychází z jazyka ADA), je hojně používán v akademické sféře. **Verilog** se hojně používá v komerční sféře, jelikož jeho syntax připomíná jazyk C. Dle dostupných diskuzí na internetových fórech je zřejmé, že toto prostředí není tak explicitní jako VHDL, ale používá se díky své rozšířenosti. **SystemVerilog** transformuje dle standardu IEEE 1800 jazyk Verilog pro účely modelování, vytváření testovacích scénářů (angl. *testbench*) a verifikaci navrhnutých struktur. **SystemC** je standardem dle IEEE 1666, který se používá pro modelování spíše v průmyslové praxi.

Nástroje pro vývoj aplikací pro FPGA prostředí nabízí více úrovní abstrakce, pomocí kterých může vývojář navrhovat systém a integrovat ho:

- Gate level – nejnižší úroveň abstrakce, kdy vývojář propojuje signály mezi

hradly. Překládá se přímo do výpisu (angl. *netlist*) propojení (angl. *bus matrix*) a nastavení logických elementů (angl. logic element).

- Register Transfer Level (RTL) – nabízí úroveň propojování kombinační logiky a sekvenční logiky. Syntetizuje se na gate level úroveň.
- TransactionLevel Modelling (TLM) – jedná se o modelování na vyšší úrovni založené na propojování komunikačních kanálů mezi funkčními bloky a definování transakcí (zpravidla pomocí SystemC).

Jelikož je implementace pokročilejších algoritmů pro FPGA, tedy v tzv. RTL logice komplexní a časově náročná činnost, lze s výhodou použít moderní metodu (HLS) generování kódu pro danou platformu z obecného předpisu ve vyšším programovacím jazyce. Pro tuto operaci je možné použít různých nástrojů a frameworků, jedním z nich je např. OpenCL (Open Computing Language), který rozšiřuje programovací jazyk C (C99) o direktivy pro určení fyzického místa a o funkce pro práci s paralelními úlohami. Dnes už je pomocí této metody možné generovat i některé periferie procesoru, což spěje k vývoji aplikace dle modelu (viz MDA).

5.3 Výběr soft-core

V této kapitole jsou uvedeny některé procesory implementované v hradlovém poli, které byly brány v potaz při výběru pro část reálné implementace této práce. S jejich popisem jsem uvedl také důležité vlastnosti. Při výběru kladu důraz na dostupnost, otevřenost, programovací jazyk zdrojových kódů (VHDL), četnost použití zjištěnou odhadem a nepříliš vysokou komplexnost. Tato práce nemá za cíl důkladně porovnat všechny dostupné soft procesory, ale pouze zvážit a vybrat nejvhodnější procesor pro účely této práce.

5.3.1 Plasma

Jedná se o open-source soft-core, který byl vyvinut nadšencem Stevem Rhoadsem [108] a který je hojně přejímán v akademickém prostředí, a to hlavně díky jeho názornosti a otevřenosti. Soft-core disponuje 32-bitovou MIPS instrukční sadou a vlastním operačním systémem reálného času (Plasma-RTOS) obsahující pouze některé knihovny, jako je např. TCP/IP stack. V akademickém prostředí existuje velká řada rozšíření tohoto procesoru počínaje zlepšením spolehlivosti až po zapojení více jader dohromady. Tato rozšíření však mnohdy nejsou k dispozici. Kvůli chybějící komerční podpoře tedy nemůže tento procesor disponovat konsistentní základnou podporovaných softwarových modulů.

5.3.2 Leon

Soft processory typu LEON jsou postaveny na *SPARC V8* architektuře a používají tedy její instrukční sadu. Jedná se o plnohodnotný škálovatelný 32-bitový soft procesor obsahující jednotku pro správu paměti, cache paměť pro instrukce i data, FPU jednotku, AHB sběrnici a jiné podpůrné subsystémy. Dosahuje výkonu až 1.4 DMIPS/MHz. Jeho licence je otevřená pro verze Leon2 a Leon3. Na těchto typech soft procesorů lze provozovat RTOS, jako je RTEMS, VxWorks, a QNX. K těmto procesorům také existují simulační nástroje (viz *grsim*) a debugovací nástroje s možností profilace programu (viz *grmon*).

Existují i upravené verze, které jsou určené do náročných prostředí, jako je letectví a náročnější procesní průmysl (řízení jaderných elektráren). Tyto verze (*Leon3-FT* a *Leon4-FT*) jsou již komerčním produktem firmy *Cobham Gaisler AB*. Tento produkt je již dodáván jako hotový čip (tzv. ASIC) a disponuje mechanismy pro detekci a korekci až 4 bitových změn (tzv. SEU) pomocí vylepšeného ECC algoritmu pro auto-korekci těchto chyb. Také je vybaven jednotkou pro detekci softwarových chyb. Tyto podpůrné subsystémy zajišťující nejvyšší možnou míru funkční bezpečnosti na úrovni procesoru nesnižují jeho výkon oproti variantám *Leon3* a *Leon4*. [28]

Za účelem vybavení procesoru novými součástmi se tento jeví jako nejvhodnější varianta. Na druhou stranu již existuje komerční řešení, které zvyšuje jeho funkční bezpečnost, a umožňuje tedy certifikaci na SIL3. Tato varianta je ovšem majetkem firmy Cobham, která tato řešení nasazuje v oblasti vesmírného průmyslu.

5.3.3 MicroBlaze

Tento soft procesor je hodně využívaný výrobci nasazujícími *Xilinx* FPGA, na který je také optimalizován. Šířka sběrnice je 32-bit a kromě ALU obsahuje také FPU, volitelně jednotku správy paměti a cache paměti pro instrukce a data. Řadič disponuje 3stupňovým nebo 5stupňovým zřetězením operací (angl. *pipelining*). Více v [142]. Ze strany operačních systémů je podporován systémy *mainline* Linux a ve variantě bez MMU systémem FreeRTOS. Tento soft procesor nebyl vybrán z důvodu licenčních podmínek.

5.3.4 PicoBlaze

Jedná se o hojně používaný 8-bitový soft procesor optimalizovaný pro *Xilinx*. Velikost programu je omezena na 1 kB kódu v assembleru. Všechny instrukce trvají dva takty. Více v [142]. Tento soft procesor nebyl vybrán z důvodu licenčních podmínek a jeho

omezených možností, jako je omezená paměť, neexistující port RTOS a slabá podpora jazyka C (existují akademická řešení implementující částečně kompilátor z C).

5.3.5 Nios II

Jedná se o vlajkový soft procesor firmy *Altera/Intel*. Je dodáván jako IP core, takže není možno bezplatně zasahovat do jeho struktury. Šírka sběrnice je 32-bit. Existuje ve dvou variantách *Nios II-e* jakožto rychlejší jednodušší varianta a *Nios II-f* jakožto plnohodnotný procesor s jednotkou MMU. Obsahuje NVIC subsystém přerušení a má vyvedenou Avalon sběrnici pro připojení dalších subsystémů. Díky jeho rozšířenosti existují porty operačních systémů reálného času, např. FreeRTOS.

5.4 Zhodnocení

FPGA obvody jsou ideální platformou pro implementaci algoritmů do hardware ve fázi výzkumu a vývoje. Pro detekci defektů, které mohou zapříčinit dokonce i zastavení běhu procesoru a kde je potřeba vysokého výpočetního výkonu hlavního subsystému, je vhodné použít pro monitorování neinvazivních (*in-situ*) technik. Monitorovací subsystém také musí disponovat dostatečným výpočetním výkonem nebo své úlohy paralelizovat, aby dokázal provádět operace co nejrychleji a aby nedocházelo k časovým anomáliím. Pro získání informací z hlavního subsystému je tedy vhodné monitorovat přímo stav procesoru, tedy jeho vnitřní registry. Pro implementaci a testování takového systému se proto jeví jako nejvýhodnější implementovat hlavní subsystém jako soft-core, kde je možné napojit se na vnitřní sběrnici procesoru, popř. jeho registry.

Za účelem reálné implementace jsem tedy pro svou práci vybral soft-core Nios a mapovanou periferii jako rozhraní. Mapovaná periferie tedy bude přijímat události ze systému s časovou značkou a dále je zpracovávat pomocí vytvořeného algoritmu.

Zvolil jsem vývojový kit DE0-Nano-Soc (firmy Altera) obsahující SoC Cyclone V (hradlové pole v kombinaci s procesorovou jednotkou architektury ARM Cortex-A9) z důvodu cenové dostupnosti. Po návrhu systému (v software Quartus) s Nios II soft-core a mapované periferie na sběrnici Avalon poskytující pouze zapisovatelné registry (viz. příloha A.3) jsem integroval uC RTOS a vyzkoušel funkčnost celé architektury. Přitom jsem narazil na nevýhodu, kdy se při změně v architektuře systému musí celé software vybavení znovu dlouho generovat, což ztěžuje prototypování a rychlé ladění. Tento přístup však také umožňuje simulaci celého systému (více v kap. 9).

6 NÁVRHOVÉ VZORY

V této kapitole vysvětlím, co jsou návrhové vzory (viz kap. 6.1) v pojetí vytváření vestavných aplikací reálného času, a začlením je do typových kategorií (viz kap. 6.3). Dále uvedu důvody, proč je používat (viz kap. 6.2), a také budu diskutovat jejich nevýhody (viz kap. 6.4). Představím metody popisu a formalizace těchto vzorů (viz kap. 6.5). Dále budu diskutovat rozdíl mezi pojmy *model*, *návrhový vzor*, *vzor architektury* a *idiom* (viz kap. 6.5.7). Poté již uvedu jednotlivé vzory, které definuje dostupná literatura (viz kap. 6.6).

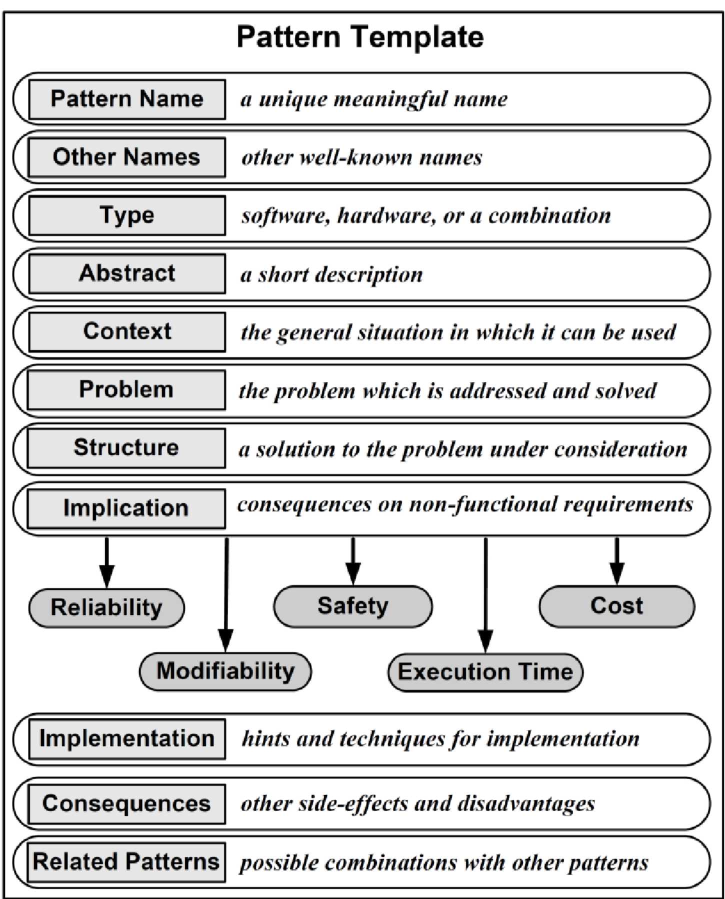
Informace potřebné pro popis a definování jednotlivých návrhových vzorů čerpám z knižní publikace [40], jejíž autor je akademik zabývající se nástroji pro popis a modelování real-time systémů pomocí návrhových vzorů. Definované vzory vycházející z komponent a chování RTOS čerpám převážně z knih [77] a [81], které se zabývaly návrhem a analýzou systémů reálného času s důrazem na jeho strukturu. Následné vytváření modelů jednotlivých uvedených vzorů zakládám na informacích uvedených v knize [53]. Kromě knižních pramenů čerpám také z konferenčních příspěvků (za váženější příspěvky pokládám např. příspěvek prof. Tempelmeira z Rosenheim univerzity [134]), technických zpráv, instruktážních videí společností (např. Intel a MathWorks) a závěrečných prací (např. [13], kde se autor zaměřuje na formální popis uvedených návrhových vzorů).

6.1 Popis

Návrhový vzor v programátorské praxi je definován jako *obecné řešení častého problému při návrhu software* [17] nebo jako *explicitně pojmenovaný obecný princip*. Přesné vyjádření definice nemá svou standardizovanou podobu, avšak mezi definicemi jednotlivých autorů nelze nalézt významnějšího rozporu. Z definice plyne jejich využití, a to jako užitný vzor při návrhu systému, komponenty a programu. Uplatní se tedy ve fázi návrhu životního cyklu vývoje software. Forma definice určitého návrhového vzoru je dána nutnými atributy, jako je název, popis řešeného problému, implementace a shrnutí možných dopadů. [5] uvádí formu definice dle obr. 6.1.

Návrhové vzory jsou již dlouhodobě využívaná metoda v programátorské praxi. Dle tvrzení autorů literárních děl uvádím první významné milníky v pokroku využití pojetí návrhových vzorů. V roce 1977 vznikla na univerzitě v Berkeley první tištěná zmínka o pojetí vzorů jako schémat pro vytváření větších celků [26]. V roce 1987 vznikla první technická zpráva o využití pojetí návrhových vzorů při vytváření počítačových programů dle objektově orientovaného paradigmatu [11]. [134] tvrdí, že použití návrhových vzorů v embedded oblasti nastalo daleko dříve než v 90. letech,

a to v roce 1984 knihou *Buhr: System design with ADA* [20]. V roce 1994 byla vydána kniha klasifikující návrhové vzory a definující vzájemné vztahy mezi nimi, které lze použít ve vyšších programovacích jazycích [50].



Obr. 6.1: Šablona pro definici návrhového vzoru podle [5]

6.2 Použití

Používání návrhových vzorů při vytváření programů podporuje správné programátorské postupy vedoucí ke spolehlivějšímu chodu programu. Zde se uplatňuje princip *použití již ověřeného*. Tímto se také sníží náklady na programátorské zdroje pro danou aplikaci a navíc dojde ke zpřehlednění kódu, který lze pak lépe dokumentovat. Tento přístup vede dle norem IEC 61508 a ISO 26262 ke zlepšení funkční bezpečnosti výsledného programu. Kategorizací a definováním vazeb lze navíc tyto vzory udržovat a snadněji používat [105].

Kvantifikovat zlepšení spolehlivosti a funkční bezpečnosti při použití návrhových vzorů je velmi obtížné. [5] navrhl metodu měření změny spolehlivosti standard-

ního systému a systému dle návrhového vzoru, přičemž použil simulační metodu Monte Carlo. Ve své práci kategorizoval některé návrhové vzory, vytvořil z nich spolehlivostní modely a vypočítal jejich výslednou spolehlivost. Dle výpočtů navrhl kvalifikaci (doporučeno či velmi doporučeno) těchto vzorů z hlediska standardu funkční bezpečnosti.

Proces vytváření programů určených pro real-time embedded prostředí se musí dle normy IEC 61508 řídit postupem zvaným *V-model*. Norma ISO 26262 navíc přidává povinnost projít celým postupem vývoje dle tohoto modelu při každé změně zadávací specifikace. Tento postup se kryje s metodou *Harmony*, která říká, že je dobré projít celým procesem vývoje software po každé testovací fázi za účelem implementace návrhových vzorů do programu [39].

Některé návrhové vzory jsou již dokonce obsaženy v programovacích paradigmatech, např. objektově orientovaný přístup zapouzdřuje data a funkce s relevantní funkcionalitou do tříd. Architekturu RTOS lze tedy také považovat za návrhový vzor, který umožňuje rozčlenit části aplikace plnící dílčí úlohy. Programátor tedy už může od RTOS různých výrobců očekávat podobnou funkcionalitu a strukturu. Dle [40] je naopak u každého programu pro real-time embedded prostředí na zvážení, zda použít některý z návrhových vzorů pro běh programu přímo na CPU, nebo využít již hotového běhového prostředí se specifickými službami, které je orientováno na plnění úloh, tedy operačního systému reálného času (RTOS).

6.3 Rozdělení

[135] rozděluje návrhové vzory do více kategorií než ostatní autoři, s čímž se ztotožňuji z důvodu větší obsáhlosti rozličných návrhových vzorů. Tyto vzory jsou:

- tvořící vzory – definují způsob a implementaci vytváření tříd a objektů,
- strukturální vzory – definují strukturu tříd implementující definované operace nad daty,
- vzory chování – definují strukturu tříd implementující určitou funkcionalitu,
- vzory architektury – definují architekturu aplikace jako celku,
- real-time vzory – definují mechanismy používané v real-time systémech.

V [40] jsou identifikovány návrhové vzory používané v real-time systémech. Tyto vzory jsou rozděleny do pěti kategorií dle funkcionality, kterou zajišťují:

- vzory přístupu k hardware,
- vzory implementace pseudo-paralelního vykonávání vláken,
- vzory správy zdrojů,
- vzory deterministického chování programu či vlákna,
- vzory pro zvýšení spolehlivosti a funkční bezpečnosti.

V [105] jsou vzory extrahovány z doporučení daných standardem IEC 61508. Navíc autor definoval vazby mezi vzory a tyto vzory kategorizoval:

- vzory pro řídicí systémy, bezpečnostní systémy a jejich kombinace,
- vzory procesu řízení rizik a redukce rizika,
- vzory pro vývoj funkčně bezpečných systémů.

6.4 Problémy

S pojmem *návrhové vzory* se v literatuře pojí problémy, které vznikají při návrhu programu právě v tomto pojetí. Níže tedy uvedu názory, které se snaží tuto problematiku shrnout.

V [17] jsou uvedeny problémy, které se autor snažil vyřešit pomocí přístupu *Layered Object Model* (popsáno v kap. 6.5). Autor uvádí tyto problémy:

- trasovatelnost – díky koncepci a z důvodu omezených možností cílového programovacího jazyka je ztracena informace o použití vzoru,
- identifikace – díky omezení detailů předávaných zpráv se ztratí odkaz na zdrojový objekt, který zprávu původně emitoval,
- znovu-použitelnost – z důvodu implementace vzoru do různých tříd je nutné daný vzor implementovat pokaždé znovu,
- režie implementace – režie implementace vzoru narůstají s daným jevem,
- programovací jazyk – zdrojový kód konvenčních objektově orientovaných jazyků není vhodný pro zachycení návrhových vzorů.

Podle [13] je problémem slabá podpora návrhových vzorů ze strany softwarových nástrojů. Autor se tedy snaží vytvořit software pro podporu implementace návrhových vzorů. Další problém autor spatřuje ve slovním, tedy vágním popisu návrhových vzorů namísto formálního popisu, přičemž zmiňuje některé snahy formálních notací z akademické sféry, z čehož ale nevzešel žádný standard. Autor si dále stěžuje na nepříliš ideální stav vzdělávání nových vývojářů stran aplikování návrhových vzorů a nepoměr knižních titulů oproti problematice modelování.

[5] uvádí jako problém návrhových vzorů pouze nežádoucí implikace použití vzoru na uživatelské požadavky netýkajících se funkce vytvářeného programu. Mezi tyto požadavky patří míra spolehlivosti, funkční bezpečnosti, doba vykonávání a náklady.

Jak uvádí [134], některé formální a semi-formální jazyky jsou nedostatečné pro popis některých návrhových vzorů. Jako příklad uvádí popis *Priority Ceiling* vzoru pomocí UML diagramu. Zde je nutné podotknout, že UML diagram slouží pro vytvoření formálního popisu struktury nebo stavu a ne procesu, takže nezachytí chování části systému.

Pro softwarovou práci s návrhovými vzory (aplikace, verifikace, automatizovaná

implementace) je tedy nutné formalizovat návrhový vzor. [10] diskutoval výhody a nevýhody formalizace návrhových vzorů. Mezi problémy, které by formalizace měla vyřešit, řadí:

- nedefinované vztahy mezi jednotlivými návrhovými vzory,
- nevyřešený způsob validace programu oproti specifikaci návrhového vzoru,
- operace se vzory v rámci automatizované tvorby programu.

Také uvádí řadu myšlenek, které svědčí proti formalizaci vzorů. Tyto myšlenky jsou výsledkem rešerše. Jako námitky formalizace tedy uvádí:

- zaměření se pouze na řešení daného problému bez ohledu na analýzu problému,
- formalizace vzoru zamezí správnému pochopení jeho podstaty [21],
- popis by měl být vágní, protože přesným popisem dané řešení nelze už nazývat *vzorem* [21],
- správně zobecněný vzor nelze explicitně vyjádřit, protože neobsahuje fixní elementy.

K těmto námitkám ale na druhou stranu uvádí důvody, kterými snižuje váhu těchto problémů:

- konkrétní specifikace řešení neznamená opomenutí dalších důležitých částí aplikace vzorů,
- nejasné a nejednoznačné formulace určitě nenapomáhají přesnému pochopení vzorů,
- specifikace může být precizní a současně zaměřená na obecné problémy,
- pokud vzor neobsahuje fixní elementy, tak neexistuje způsob, jak vzory popsat a pochopit.

Z diskuze vyplývá, že by se pro popis měl použít nástroj, který je zároveň obecný na určité úrovni abstrakce a zároveň přesně popisuje daný návrhový vzor. Návrhové vzory jsou dle definice popisovány slovním popisem, avšak tlak vývoje vhodných nástrojů automatizovaného inženýrství zapříčiňuje vznik přístupů přinášející různé způsoby a pohledy na takový vhodný nástroj. Z myšlenek uvedených v této kapitole vyplývá, že požadavky kladené na formální popis spočívají v přesné interpretaci vzoru a zároveň jsou v rozporu s obecným přístupem. Kompromisem tedy je formální popis na určité úrovni abstrakce, který disponuje vlastnostmi, jako je trasovatelnost, identifikace a znovu-použitelnost.

6.5 Metody notace a formalizace

Dostupná literatura nabízí různé druhy notace a pokusy o formalizaci zápisu návrhových vzorů. Některé vycházejí z programovacích jazyků, některé jsou pouze grafické a slovní, jiné staví na principech matematického aparátu (např. predikátové logiky).

Pokusy o notaci a formalizaci jsou uvedeny níže (čerpáno převážně z přehledu v [10] a [13]).

6.5.1 LayOM

Layered Object Model se snaží jednotlivé funkcionality implementovat jako vrstvy objektu. Jedná se o meta-jazyk, který dokáže vyjádřit GoF vzory a má blízko k implementaci, ale nemá ambice na abstraktní popis vzorů a nepřináší víc než UML. Dle [13] sám autor říká, že tento nástroj je do značné míry překonanou záležitostí.

6.5.2 Atributy

Jedná se o zachycení vzorů v programovacím jazyce pomocí atributů řešící problém trasovatelnosti vzorů. O této možnosti se zmiňuje již [62] a dnešní platforma .NET již tyto atributy podporuje, a dokonce s nimi její CASE nástroje umějí pracovat. Nevýhodou zůstává neúplný popis vzorů plynoucí ze své podstaty.

6.5.3 LePUS

Language for Patterns Uniform Specification je deklarativní jazyk vyššího řádu používající jednočlennou logiku, který kromě matematické notace umožňuje i grafický zápis. Autor také v [45] poukazuje na problémy ostatních notací, jako je *LayOM*, *Contracts*, *Constraints* a *DisCo*. *LePUS* lze údajně transformovat do logického jazyka PROLONG a pomocí automatizovaných nástrojů implementovat. Nevýhodou tohoto přístupu ale zůstává horší trasovatelnost ve zdrojovém kódu a horší srozumitelnost pro programátory.

6.5.4 Jazyk typu Z

Tento typ formalizačního jazyka uvedený v [124] jako nástroj použitelný při navrhování programu je postaven na predikátové logice a matematicky definuje programové operace. Tento jazyk je vhodný pro formální popis programu, i když je méně čitelný pro programátora. Autor už ale nerozebírá možnost generování vlastního kódu programu.

6.5.5 UML

Tato objektová notace se dle dostupných publikací nejčastěji používá pro popis vzorů, protože je srozumitelná i pro programátory a používá i grafické znázornění. UML definuje objekty, jejich strukturu a vztahy mezi nimi. Dle akademiků bádajících v této

oblasti je tento popis vzorů pro real-time aplikace nejednoznačný a neobsahuje potřebné výrazové prostředky, a proto navrhovali vylepšení (např. [86] nebo [37]). Jedno z posledních vylepšení [18] se nazývá *UML-MARTE* a zakládá se na jazyku *Maude*. Tyto snahy o přesný popis aplikace právě pomocí UML jazyka jsou podporovány výrobci tzv. CASE nástrojů, které podrobněji uvedu v kap. 6.5.6.

Dle mého názoru je UML dobrým nástrojem pro rychlé uvedení do problematiky daného návrhového vzoru, k čemuž pomáhájí již existující rozšíření pro použití v *real-time embedded* prostředí. V současné době již existují nástroje pro správu a aplikaci návrhových vzorů (viz kap. 6.5.6), avšak nejsou vhodné jako prostředek pro automatizovanou implementaci vzoru ve své standardní verzi. Postupným vývojem CASE nástrojů a specifikace jazyka UML i pro *real-time embedded* prostředí se tento způsob stává v této oblasti používanějším než dříve. Formální notace (např. LePUS nebo jazyk Z) jsou zase vhodnější pro formální verifikaci a validaci. Důsledkem tedy je nejednotnost nástroje pro notaci návrhových vzorů a větší benevolence v aplikaci návrhových vzorů.

6.5.6 Nástroje pro práci s návrhovými vzory

Pro práci (skládání, údržba, verifikace nebo implementace) s návrhovými vzory slouží software typu tzv. CASE nástrojů. Může se jednat o primární nástroje (např. DPA-Toolkit [42]) nebo o doplňkovou funkci komplexnějších nástrojů (např. Enterprise Architect [131]). Dnes už práci se vzory dokonce podporují některá programovací prostředí (např. Eclipse a Visual Studio). Uvedené nástroje používají jako modelovací jazyk UML i přes jeho nevýhody, a to právě z důvodu snadné čitelnosti programátory. Do Eclipse jakožto univerzálního prostředí mohou být implementovány pluginy OSATE a Ocarrina.

6.5.7 Model vs. návrhový vzor vs. vzor architektury vs. idiom

Model je matematicky vyjádřená skutečnost na určité úrovni abstrakce. Metodika modelem řízená architektura (MDA) a návrh založený na modelu používají model jako nosný nástroj pro vyjádření, dokumentaci a předmět testování při návrhu systému. Průchodem vývojovými kroky systému se tento model tvoří, zpřesňuje, testuje, či dokonce implementuje. Jedná se tedy o abtraktní nástroj upravený v kontextu daného problému. Model by měl být ideálně nezávislý na implementačním programovacím jazyku a měl by umožňovat bezkolizní implementaci.

Návrhový vzor (angl. *design pattern*) v softwarovém inženýrství představuje obecné řešení daného problému (viz. kap. 6.1). Nejedná se tedy o ohraničenou komponentu, jako je např. knihovna nebo úsek zdrojového kódu obsahující algoritmus.

Bruce Douglass definuje návrhové vzory jako: „Design patterns are generalized recurring optimization problems. Design patterns are reified structures of object collaboration that reappear in a variety of contexts.“ [41] Tedy ve volném překladu říká, že návrhové vzory jsou zobecněné opakující se řešení. Jsou to odzkoušené struktury objektů, které se objevují v různých kontextech. Autor také specifikuje způsob použití návrhových vzorů pro vytvoření prvotního modelu systému. Tento model může být pak dále zpřesňován pro danou aplikaci.

Frank Buschmann v [21] vymezuje pojem *idiom*: „Idioms deal with the implementation of particular design issues. An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language. Idioms represent the lowest-level patterns. They address aspects of both design and implementation.“ Tedy ve volném překladu říká, že *idiom* je vzor na nejnižší úrovni a zabývá se konkrétní implementací návrhu v daném programovacím jazyce. *Idiom* tedy popisuje, jak implementovat konkrétní aspekty komponent nebo vztahy mezi nimi pomocí vlastností daného jazyka.

Návrhové vzory jsou platné a používají se v dané doméně. Naproti tomu vzor architektury přináší obecné řešení napříč doménami (např. hardware a software) systému, tedy v rámci jeho architektury. V procesu návrhu aplikace se tedy jedná o návrh systému na nejvyšší úrovni, kde se tyto vzory uplatňují. Tuto kategorizaci, resp. členění na vzory architektury, návrhové vzory a idiomy navrhuje také [150]. V pojetí informačních technologií je tedy vhodné uvést termíny návrhový vzor a vzor architektury do souladu s životním cyklem software dle standardu, např. IEC61508 nebo ISO26262 (viz obr. v příloze A.2). Mnohé publikace z oblasti informačních technologií nerozlišují mezi těmito dvěma pojmy a autoři používají jen termín návrhový vzor (angl. *design pattern*).

6.6 Návrhové vzory

Zde stručně uvedu existující relevantní návrhové vzory pro vývoj *real-time* aplikací v *embedded* prostředí. Relevantní vzory čerpám z knihy [40] a ze série knih Pattern Oriented Software Architecture, které se ale spíše zabývají návrhovými vzory pro programovací jazyky. Pro účely této práce omezím počet zkoumaných návrhových vzorů na ty, které lze použít pro návrh, resp. implementaci *real-time embedded* programu. Jedná se tedy o vzory, které definují dílčí funkcionalitu. Dle kapitoly 6.3 se jedná o vzory v kategoriích přístup k hardware, implementace pseudo-paralelního vykonávání vláken, správy zdrojů, deterministického chování vlákna a některé vzory pro zvýšení spolehlivosti a funkční bezpečnosti.

6.6.1 Vzory architektury systému

Channel pattern specifikuje architekturu programu nebo jeho části jako sériové spojení dílčích úloh (např. zpracování vstupů, vykonání algoritmu, nastavení výstupů). **Redundancy pattern** specifikuje architekturu systému nebo programu jako více paralelně běžících subsystémů s porovnáním jejich výstupů (např. lock-step, TMR).

6.6.2 Vzory v paralelně vykonávaných systémech

V paralelně a pseudo-paralelně vykonávaných systémech je nutné řešit problémy ohledně sdílení dat a zdrojů. **Message queuing pattern** řeší způsob uchovávání požadavků (dat) pomocí fronty zpracovávající vlákna. **Guarded cell pattern** je způsob přístupu ke zdroji (datům) z více vláken za použití synchronizačního primitiva typu semafor. **Rendezvous pattern** specifikuje konstrukci zpětného volání funkce po ukončení asynchronní operace volané funkce.

6.6.3 Vzory obsluhy paměťového prostoru

Pooled allocation pattern je způsob přiřazování malých paměťových bloků vláknům na jejich žádost. **Fixed-size buffer pattern** řeší alokování paměti ve větších fixních blocích, které pokryjí požadavek. **Smart pointer pattern** řeší alokaci paměti pomocí *pointerů*, kterými přiřazuje danou oblast vláknu.

6.6.4 Vzory distribuce dat

Observer pattern je způsob datové výměny, kdy server pošle nová data klientům, které má uložené v listu namísto toho, aby se klienti museli dotazovat. **Proxy pattern** navrhuje architekturu komunikace tak, že server je odstíněn od klientů, a tedy je možné jej dynamicky realokovat.

6.6.5 Vzory paralelního běhu úloh

Active object (Asynchronní vzor) odděluje spouštění metod od provádění metod, přičemž spouštění metod může být ve svém vlastním vlákně. Cílem je přidat souběžnost použitím asynchronních volání metod a plánovače, který obsluhuje požadavky. **Event-based asynchronous** řeší problémy s Asynchronním vzorem, které nastávají ve vícevláknových programech. **Balking** je softwarovým vzorem, který na objektu vykoná nějakou akci, pouze pokud se objekt nachází v určitém stavu. **Double checked Locking** je také znám jako „optimalizace zamykání s dvojnásobnou kontrolou“. Návrh je vytvořen tak, aby zredukoval zbytečné náklady na získávání zamčení tím, že nejdříve otestuje kritérium pro zamčení nezabezpečeným způsobem (tzv. *lock hint*).

Pouze pokud uspěje, pak se opravdu zamkne. Tento vzor může být nebezpečný, pokud je implementován v některých kombinacích programovacích jazyků a hardwaru. Proto je někdy považován také za proti-vzor. **Guarded** obstarává operace, které požadují uzamčení a navíc mají nějakou podmínku, která musí být splněna předtím, než může být operace provedena. **Monitor object** je přístup k synchronizaci dvou nebo více počítačových úloh, které používají sdílené zdroje, zpravidla hardwarové zařízení nebo sadu proměnných. **Read write lock** (RWL) umožňuje souběžný přístup k objektu pro čtení, ale vyžaduje exkluzivní přístup pro zápis. **Scheduler** se používá pro explicitní kontrolu, kdy mohou vlákna vyvolávat jednovláknový kód. **Thread pool** implementuje sadu vláken pro řešení nějakého množství úloh, které jsou organizovány ve frontě. Zpravidla je výrazně méně úloh než vláken. **Thread-specific storage** je programovací metoda, která používá statickou nebo globální paměť lokálně pro vlákno. **Reactor** se používá pro vyřizování požadavků na službu, které jsou z jednoho nebo více vstupů doručovány správci služeb. Správce služeb rozdělí příchozí požadavky a přidělí je synchronně přidruženým vyřizovačům požadavků.

6.7 Zhodnocení

Průzkum literatury v oblasti návrhových vzorů vede k závěru, že ač tato metoda může být s výhodou využívána v oblasti vestavných systémů reálného času a tvorby jejich softwarového vybavení, existuje nepoměrně malé množství literatury, výzkumu a zpráv o použití. Situace se lepší v oblasti návrhu programu zvláště ve vyšších programovacích jazycích. Tato metoda tvoří synergický efekt s metodikami modelem řízená architektura a návrh založený na modelu. Také z hlediska funkční bezpečnosti je výhodnější používat ověřené postupy, principy, struktury a komponenty. Zlepšení použitím návrhových vzorů je ale obtížné kvantifikovat.

7 MĚŘENÍ REÁLNÝCH SCÉNÁŘŮ

V této kapitole se zabývám měřením, resp. záznamem událostí RTOS na reálné platformě. V kap. 7.1 uvedu všechny relevantní nástroje, které bych mohl použít pro platformu STM32F0DISCOVERY (procesor založen na architektuře ARM Cortex-M0), kterou mám k dispozici. Dále představím celý měřicí systém (viz. kap. 7.2), a potom specifikuji scénáře situací (viz. kap. 7.3), u kterých uvedu jejich popis a vlastní měření, resp. záznam událostí RTOS. Poznatky uvedené v této kapitole vychází z kap. 2.1.

7.1 Prostředí a nástroje

Pro monitorování událostí běžícího RTOS disponují jednotliví výrobci programovacích prostředí a procesorů různými nástroji. Jelikož se ale jedná o pokročilou funkci, jsou tyto pokročilé nástroje většinou zpoplatněny v závislosti na komplexnitě monitorovaných informací. Standardní software pro debugování sice zobrazují aktuálně vykonávanou instrukci, hodnoty proměnných a registrů, avšak neumožňují zobrazit záznam vykonaných funkcí. A to i přesto, že daný mikroprocesor tvorbu tohoto záznamu podporuje, např. díky ARM CoreSight koprocesoru (více v kap. 5.1.1). Tato kapitola má za úkol přinést poznatky z oblasti monitorovacích nástrojů procesorů, které jsem získal v procesu hledání a zkoušení vhodného relativně dostupného nástroje.

ST-LINK slouží jako programovací a debugovací nástroj pro procesory firmy ST. Kromě klasického externího zařízení je možné použít zabudovaný programátor ve vývojových kitech STM DISCOVERY nebo NUCLEO. Tento nástroj obsahuje ve vyšších verzích i virtuální COM port použitelný pro odesílaní debugovacích informací. Jako GDB server slouží standardní software *st-util* nebo je možné použít jiných volně dostupných, např. OpenOCD nebo pyOCD. Ve standardní konfiguraci neumožňuje vyčítat trasovací buffery procesoru (ETM ani ITM).

J-LINK (firmy IAR) slouží jako programovací a debugovací nástroj pro různé architektury procesorů. Kromě klasického externího zařízení je možné přehrát zabudovaný programátor ve vývojových kitech STM DISCOVERY. Tento nástroj obsahuje virtuální COM port použitelný pro odesílaní debugovacích informací. Jako GDB server slouží standardní software *JLink Commander* firmy SEGGER nebo je možné použít jiných volně dostupných, např. OpenOCD nebo pyOCD. Ve standardní konfiguraci neumožňuje vyčítat trasovací buffery procesoru (ETM ani ITM).

J-TRACE (firmy IAR) slouží jako programovací, debugovací a monitorovací externí zařízení pro různé architektury procesorů. Tento nástroj patří mezi nejméně

invazivní a ve standardní konfiguraci umožňuje vyčítat trasovací buffery procesoru (ETM a ITM). Je podporován různými IDE, např. KEIL uVision a IAR Embedded workbench. Jedná se o velmi drahý nástroj, který jsem pro mou práci nemohl využít.

VisualGDB je komerční plnohodnotné IDE postavené na platformě Visual Studio. Umožňuje navíc díky svým rozšiřujícím balíčkům přidat monitorovací subsystém (ProfilerRTOS) do aplikace, a tím monitorovat události RTOS skrz debugovací zařízení, které pak zobrazí v monitorovacím okně debugovacího prostředí. Operace se záznamem z monitorování není ještě na přívětivé úrovni, ale pro prvotní náhled na děje v systému dostačuje. Konfigurace celého procesu je intuitivní, a to hlavně díky snadné integrovatelnosti tohoto nástroje.

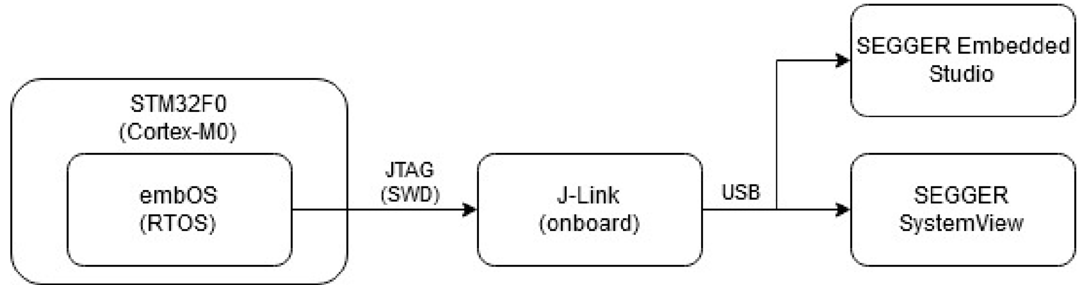
Tracealyzer (firmy Percepio) je komerční software pro zobrazování záznamu monitorování různých RTOS. Staví na propracovaném uživatelsky přívětivém prostředí, které je možné připojit na monitorovací subsystém (Trace Recorder Library pomocí COM portu, a dokonce subsystémů jiných výrobců, např. SEGGER RTT). Konfigurace tohoto nástroje je obtížná, avšak věřím, že s placenou podporou bude proveditelná.

SEGGER SystemView je volně dostupné software pro nekomerční účely, který spolu s dalšími produkty firmy SEGGER, např. SEGGER Embedded Studio, embOS a vzorovými projekty pro různá IDE tvoří rychle použitelný ekosystém. Monitorovací subsystém SEGGER RTT se v řetězci spolu s J-LINK a SystemView jeví jako přehledný nástroj pro monitorování událostí, času vykonání a měření frekvence událostí RTOS. Tento software disponuje solidními možnostmi práce se záznamem, jako je hledání dalšího výskytu, zobrazení v grafu a zobrazení posloupnosti samotných událostí s měřenými parametry. Oproti VisualGDB zahlásí problém s přeplněným bufferem, sloužícím pro přenos monitorovaných událostí (z důvodu pomalé rychlosti nebo enormního přepínání událostí), a dané problematické oblasti v záznamu vyznačí.

7.2 Měřicí systém

Jako platformu pro mé měření jsem zvolil vývojový kit STM32F0DISCOVERY, který zapouzdřuje procesor STM32F051 (taktovaný pomocí PLL na 48 MHz) postavený na architektuře ARM Cortex-M0. Kromě procesoru obsahuje vývojový kit programovací subsystém ST-LINK, který jsem přehrál na J-LINK, a standardní procesorovou bižuterii obohacenou o dvě indikační LED diody. Obešel jsem se tedy bez externího debugovacího zařízení. Jako IDE jsem použil SEGGER Embedded Studio for ARM a pro zobrazení záznamu událostí RTOS jsem použil SEGGER SystemView (viz blokový diagram 7.1).

Aplikace pro testovací scénáře (viz kap. 7.3) jsem vytvářel pro embOS, což je



Obr. 7.1: Blokový diagram měřicího řetězce

RTOS od firmy SEGGER, který může být obohacen o další knihovny pro obsluhu periferií, podporu šifrování a webového serveru. Jako monitorovací subsystém integrovaný do aplikace jsem použil vhodný SEGGER RTT. embOS zapouzdřuje pro uživatelské aplikace vrstva CMSIS-RTOS, která definuje rozhraní RTOS. Volání funkcí této vrstvy pak zaznamenává monitorovací subsystém.

7.3 Testovací scénáře

V této kapitole specifikuji scénáře pro testování chyb vykonávání RTOS z hlediska správného vykonávání úloh. V literatuře jsem nenašel žádné ustálené definované scénáře na softwarové úrovni, na kterých by se testovaly algoritmy funkční bezpečnosti, protože každý autor si definuje vlastní scénáře v závislosti na typu detekovaných chyb. Ustálené jsou pouze projevy chyb plánování, a to *deadlock*, *livelock*, *starvation* a *race condition*. Zároveň se také jedná o chyby, které jsem schopen na softwarové úrovni injektovat.

7.3.1 Deadlock

Deadlock je stav, kdy se skupina vláken blokuje navzájem pomocí synchronizačních objektů, přičemž se nevykonává žádný kód těchto vláken. Tento stav tedy zabraňuje správnému odvíjení chodu aplikace v čase.

Tuto chybu tedy simuluji pomocí dvou vláken a dvou *mutexů*, kdy vlákno nejdříve získá první mutex, a potom bude chtít v blokujícím režimu druhý, přičemž druhé vlákno už tento *mutex* získalo a čeká na první mutex. Kód scénáře je uveden v příloze A.4. Vlákna v systému tedy jsou:

- 1. vlákno („HPTask“) má prioritu 100, periodu 50 *ms* a přepne LED0,
- 2. vlákno („LPTask“) má prioritu 50, periodu 50 *ms* a přepne LED1.

Z měření událostí (viz příloha A.5) vyplývá, že ani jedno z vláken se nevykoná, protože se navzájem blokují. LED diody tedy nezmění ani jednou svůj stav. V tomto

případě by reagoval i *watchdog*, avšak v případě pokročilejší integrace do systému, resp. při vyžádání informace o vykonání vlákna. Kontrolou maximálního času vykonání událostí lze tento stav detekovat.

7.3.2 Livelock

Livelock je stav, kdy se skupina vláken blokuje navzájem pomocí synchronizačních objektů, přičemž nastalo zamezení vykonání užitečného kódu některého z těchto vláken. *Livelock* se od *deadlock* liší tím, že vlákna se snaží reagovat na nastalou situaci a nejsou stále v blokovaném stavu. Složitější detekce nastává v případech, kdy *livelock* podmínka je naplněna několikanásobným restartem systému z důvodu permanentní chyby v jednom místě. V tomto případě musí reagovat nadřazený klasifikátor (FCCU).

Jako *livelock* mohou být klasifikovány tyto stavy systému:

- *starvation* – důsledek blokace vlákna z důvodu dlouhodobě uzamknutého zdroje,
- nekonečná smyčka (angl. *infinite loop*) – proces vykonávání vlákna je uzavřen v nekonečné smyčce,
- porušení *liveness* podmínky – ostatní případy, kdy se aplikace dostane do stavu, ze kterého nevede žádané východisko.

Tento scénář se sestává ze dvou vláken a dvou *mutexů*, kdy vlákno nejdříve získá první mutex a potom bude chtít v neblokujícím režimu druhý, přičemž druhé vlákno už tento *mutex* získalo a čeká na první mutex. Kód scénáře je uveden v příloze A.6. Vlákna v systému tedy jsou:

- 1. vlákno („HPTask“) má prioritu 100, přepne LED0 a simuluje výpočet,
- 2. vlákno („LPTask“) má prioritu 50, přepne LED1 a simuluje výpočet.

Z měření událostí (viz příloha A.7) vyplývá, že ani jedno z vláken se nevykoná, protože se navzájem blokují, přičemž na pozadí RTOS produkuje činnost, kdy se snaží vyhovět synchronizačním podmínkám. LED diody tedy nezmění ani jednou svůj stav. V tomto případě by reagoval i *watchdog*, avšak v případě pokročilejší integrace do systému, resp. při vyžádání informace o vykonání vlákna. Kontrolou maximálního času vykonání událostí lze tento stav detekovat.

7.3.3 Starvation

Starvation nastává u vlákna v systému v případě, kdy některé vlákno blokuje zdroj často na dlouhou dobu, přičemž jiné vlákno potřebuje také častý přístup. Nemusí se jednat nutně o chybu v návrhu, ale tento stav může nastat např. při defektu v hardware, kterému najednou dlouho trvá odpovědět na žádost. Jedná se také o případy, kdy se některé vlákno plní nebo vybírá frontu příliš rychle, přičemž jiná

vlákna musí na dané zdroje čekat příliš dlouho. Situace, že vlákno je plánovačem přehlíženo, i když je v připraveném stavu, může nastat hlavně při vyšší výpočetní zátěži za podmínky nepříliš optimálního návrhu.

Tuto chybu simuluji pomocí dvou vláken a dvou *mutexů*, kdy vlákno nejdříve získá první mutex a potom bude chtít v neblokujícím režimu druhý, přičemž druhé vlákno už tento *mutex* získalo a čeká na první mutex. Kód scénáře je uveden v příloze A.8. Vlákna v systému tedy jsou:

- 1. vlákno („HPTask“) má prioritu 100, periodu 50 ms a přepne LED0,
- 2. vlákno („LPTask“) má prioritu 50, periodu 50 ms a přepne LED1.

Z měření událostí (viz příloha A.9) vyplývá, že druhé vlákno se nevykoná celé. Pouze se snaží získat oba *mutex*, ale druhý *mutex* se mu kvůli blokaci prvního vlákna nepodaří získat, takže uvolní první *mutex* a pokusí se o celou rutinu znovu. První vlákno se vykonává pravidelně celé. První LED dioda pravidelně bliká, ale druhá nemění svůj stav, resp. kód v synchronizační podmínce se nikdy nevykoná. V tomto případě by reagoval i *watchdog* za předpokladu, že by byla použita pokročilá metoda integrace do systému vyžadující informace o úspěšném vykonání všech možných cest v daných vláknech. Kontrolou posloupnosti událostí a příp. i jejich návratových hodnot lze tento stav detekovat.

7.3.4 Race condition

Race condition je stav, kdy více vláken soupeří o stejný zdroj ve stejném čase se stejnou prioritou nebo s různou prioritou, přičemž se navzájem přetahují o tyto zdroje.

Pro tento scénář jsem zvolil vlákna s různými prioritami, ale navzájem se přetahující o *mutex* díky neblokujícímu režimu a simulaci práce vlákna systémovým zpožděním. Kód scénáře je uveden v příloze A.10. Vlákna v systému tedy jsou:

- 1. vlákno („HPTask“) má prioritu 100, přepne LED0 a simuluje periodu 50 ms,
- 2. vlákno („LPTask“) má prioritu 50, přepne LED1 a simuluje periodu 100 ms.

Dle kódu může implementovaná chyba být klasifikována jako *livelock*, který se systém snaží řešit (díky implementaci časovače), z čehož vznikne série náhodných pokusů o přístup k jednomu zdroji. Z měření událostí (viz příloha A.11) vyplývá, že vlákna tedy vykonají svůj kód celý, avšak jejich periody jsou náhodné a RTOS se potýká s častým a rychlým voláním svých funkcí. V tomto případě některé oblasti nejsou zaznamenány, protože použitý měřicí systém má svoje limity (viz. kap. 7.2), a jsou tedy podbarveny červeně. Z měření period měřicím systémem je patrná neshoda měřené a požadované frekvence vykonávání vláken. V tomto případě by *watchdog* pravděpodobně nezareagoval, protože vlákna jsou vykonána celá. Kontrolou posloupnosti událostí a příp. i jejich návratových hodnot lze tento stav detekovat.

7.4 Zhodnocení

Zde uvedená měření dávají náhled na problémy provozu RTOS z hlediska plánování úloh a jiných chyb, které se tímto projeví. Z uvedených scénářů je patrné, že systém má rozličné reakce na tyto chyby z hlediska frekvence volání funkcí RTOS a že po dlouhodobém zkoumání a měření je dokonce možné kategorizovat tyto projevy vůči příslušným chybám. Vyplývá také, že pokud požadujeme plné monitorování, je nutné, aby monitorovací subsystém byl co nejméně závislý na monitorovaném systému. Na základě těchto projevů lze také říci, jak budou reagovat jednotlivé implementace algoritmů funkční bezpečnosti. Vědecké měření a vyhodnocování těchto kombinací by určitě přineslo mnoho dalších cenných poznatků a dalo by argumentační základ zkušenostem z programátorské praxe. Také důkladné měření projevů jednotlivých chyb (v software, nebo dokonce v hardware) přinese data pro kvantifikaci spolehlivosti.

Výše uvedené scénáře tedy mohou v této či pozměněné podobě sloužit jako testovací scénáře pro vyhodnocení diagnostického pokrytí nových technik funkční bezpečnosti. Pro mou práci je dostačující výstup, že monitorováním správné posloupnosti (v doméně místa a času) událostí RTOS je možné detekovat chyby, které nejsou zcela pokryty jinými technikami.

8 ONLINE KONTROLNÍ SYSTÉM

Tato kapitola se zabývá návrhem kontrolního systému, což je jeden z cílů práce. Z obecných cílů uvedených v kap. 1 práce jsem cíle své práce díky provedeným rešerším konkretizoval a definoval si užší meze, ve kterých mám v plánu těchto cílů dosáhnout (viz kap. 8.1). Díky tomu mohu uvést popis online kontrolního systému jako celku (viz kap. 8.2). Potom přejdu k volbě modelovacího nástroje sledovaného systému a k analýze dílčích aspektů tohoto nástroje z hlediska požadavků a příslušné teorie. Poté už definuji navrhovaný model a jeho chování pomocí formálního popisu (viz kap. 8.3). Dále uvedu jeho implementaci v FPGA (viz kap. 8.3.6) a nastíním proces tvoření modelu (viz kap. 8.3.7). Nakonec se pokusím o shrnující definici tohoto a podobných systémů ve formě vzoru architektury (viz kap. 8.5).

V celé kapitole tedy budu popisovat a definovat kontrolní systém s ohledem na jeho reálnou implementaci. Bude se tedy jednat o formální popis konkrétního systému. Naopak v poslední podkapitole se pokusím o definici návrhového vzoru, jakožto obecného popisu řešení problému online kontroly běhu programu dle dostupného modelu.

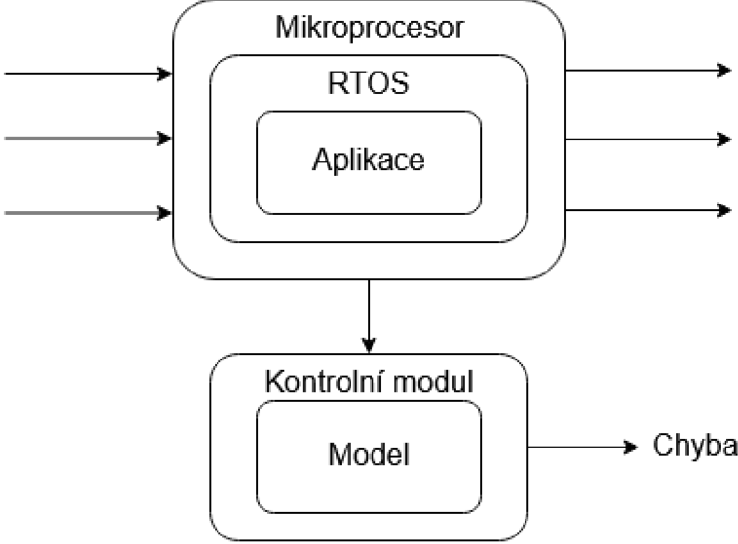
8.1 Cíle online kontrolního systému

Zde uvedené požadavky na návrh online kontrolního subsystému vyplývají z provedených rešerší zaznamenaných v předchozích kapitolách. Za účelem potlačení *common cause faults* musí dané řešení být implementovatelné v hardware, resp. v hradlovém poli pro snadnou možnost zlepšování. Subsystém monitoru by měl co nejméně zatěžovat sledovaný subsystém, aby sledovací rozhraní co nejméně rušilo běh aplikací a zároveň aby monitor byl schopen správného chodu v případě poruchy. Navrhovaný model musí být podporován širokým spektrem nástrojů a umožňovat formální verifikaci. Tento model také musí být snadno získatelný, pokud použijeme metodu získání modelu z již existujícího kódu, a také musí umožňovat generování kódu aplikace nebo její struktury v případě procesu implementace z modelu. Model, resp. online kontrola, bude vycházet z konstruktu vyhodnocení časových odchylek a odchylek vykonávání programu plynoucí z determinismu jako vlastnosti RTOS.

8.2 Popis online kontrolního systému

Navrhovaný online kontrolní systém se skládá z **modulu rozhraní** k danému procesoru, pomocí kterého je možné získat v reálném čase data o běhu systému. Tato data dále vstupují do hlavní části kontrolního systému, a to je **kontrolní modul**,

který vyhodnotí chybu běhu programu a její povahu. Výstupem kontrolního modulu jsou již informace o chybě a jejím druhu. Blokové schéma systému je zaznamenáno na obrázku 8.1.



Obr. 8.1: Architektura systému online kontroly systému

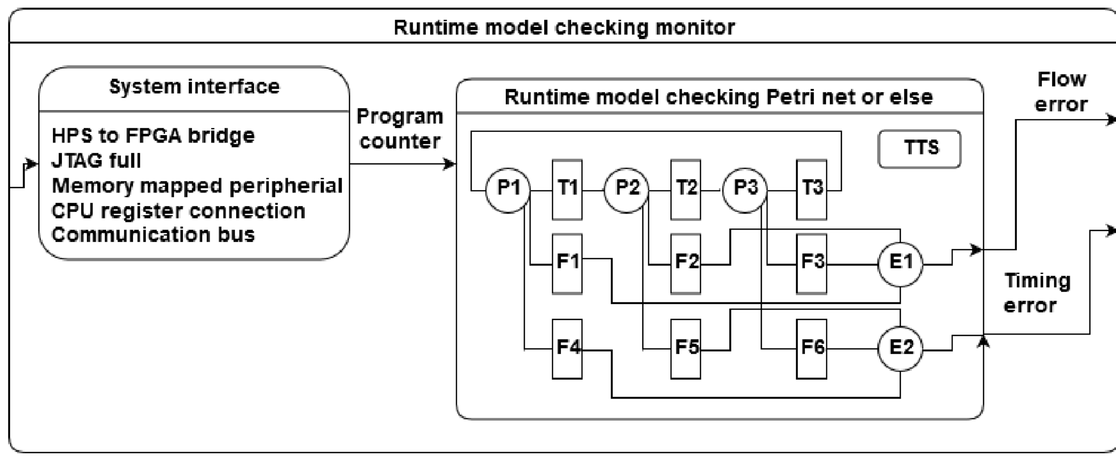
8.2.1 Modul rozhraní

Základním předpokladem pro kontrolu běhu programu je získání adekvátních informací o daném systému. Techniky pro sběr těchto informací se dělí z hlediska integrace do monitorovaného systému na invazivní a neinvazivní (viz kap. 5.1). Hypoteticky můžeme získat všechny informace o systému, jako jsou jeho vnitřní registry (např. *Program counter*). Dále je možné napojit se na vnitřní sběrnici procesoru a odposlouchávat jednotlivé zprávy (viz projekt [133]). Pro získání událostí operačního systému bude postačovat implementovat periferní jednotku na jednu z vyvedených sběrnic a monitorovaný program upravit o instrukci zápisu dané události v daném čase (více o této implementaci v kap. 5.1).

8.2.2 Kontrolní modul

Jádrem navrhovaného systému je kontrolní modul. Kontrolní modul má za úkol zpracovat příchozí data o běhu systému v reálném čase a vyhodnotit, zda jsou data validní právě vzhledem k modelu a aktuálnímu stavu sledovaného systému. V případě nesrovnalosti mezi modelovaným chodem programu a jeho skutečným chodem vyhodnotí kontrolní modul chybu a signalizuje ji pomocí výstupů. Ústředním

prvkem kontrolního modulu je tedy model sledovaného systému na určité úrovni abstrakce. Právě v tomto konstruktu spočívá idea kontroly běhu programu za jeho chodu. Blokové schéma tohoto principu shrnuje obr. 8.2.



Obr. 8.2: Blokové schéma online kontrolního systému

8.2.3 Model sledovaného systému

Sledovaný embedded systém se skládá z procesoru definovaného architekturou a programovým vybavením. Dle kap. 8.1 uvažuji architekturu programového vybavení skládající se z operačního systému reálného času a uživatelského programu.

Specifikace modelu sledovaného systému určuje obor chyb, který je možné daným online kontrolním systémem detekovat. Mimo přístupy uvedené v kap. 3, jimiž jsou monitorování toku programu (viz [27]) a kontrola řídicího systému pomocí modelu soustavy, navrhuji níže uvedené čtyři modely:

- Model událostí operačního systému reálného času,
- model chybových vzorů,
- model profilace běhu programu,
- model včasného vykonání vláken programu.

V následujících podkapitolách jsou tyto návrhy blíže specifikovány. Z těchto možností jsem pro splnění cílů práce vybral model událostí operačního systému reálného času, který dále formálně definuji a důkladně analyzuji.

Model událostí operačního systému reálného času

Tento model má obecně formu orientovaného grafu, přičemž hranám, které je možno chápat jako přechodové podmínky, přiřazuje události systému, které větví jeho chod z hlediska operačního systému. Definuje tedy možné cesty vykonávání programu

v závislosti na jeho aktuálním stavu. Aktuální stav je v tomto případě definován událostmi, které v operačním systému vyvstaly, a časem, který mezi těmito událostmi uplynul. Kontrolní modul tedy sleduje čas jednotlivých úseků programu a jejich správné pořadí. V případě nesprávného pořadí vykonávání úseků programu či v případě výrazné odchylky jejich času vykonání vyhodnotí kontrolní modul chybu a signalizuje ji. Předpokladem je, že model je formálně zkontrolován offline a tedy neobsahuje *deadlock*, *livelock* ani *race condition*, což by tedy měl být schopen mimo jiné detekovat. Nevýhodou je, že model nezahrnuje aktuální data uživatelského programu, a tedy nemůže kontrolovat správné větvení.

Z modulu rozhraní je tedy nutné získat minimálně následující informace o systému:

- Výskyt události,
- typ události,
- doba trvání události.

Druhá možnost je získat informaci o vykonávání daného úseku programu z registru Program Counter, což patří mezi neinvazivní techniky. Navíc díky této informaci bude model přesně monitorovat aktuální pozici programu oproti estimaci pozice programu v případě redukovaných informací (událostí systému).

Model chybových vzorů

Model chybových vzorů je založen na apriorní znalosti sledu událostí systému, které vedou k chybovému stavu. Tyto různé vzory sledů událostí je potom možné modelovat jako konečné stavové automaty (angl. Finite State Automaton) a implementovat jako standardní stavové automaty. Kontrolní modul tedy občerstvuje model dle aktuálních informací z monitorovaného systému a v případě dosažení konečného stavu jednoho z automatů reaguje na nastalou situaci. Předpokladem je, že model obsahuje všechny možné sekvence vedoucí k chybovému stavu a že modelované sekvence nekopírují sekvence normálního vykonávání programu.

Z modulu rozhraní je tedy nutno získat minimálně následující informace o systému:

- Výskyt události,
- typ události.

Model profilace běhu programu

Tento model může mít formu tabulky, kde klíčem jsou definované úseky programu a hodnotami jsou časy vykonání těchto úseků, a to formou intervalu. Za běhu programu by byly měřeny aktuální časy vykonání těchto úseků a porovnávány s daným intervalem. V případě nesrovnalosti je detekována chyba. Tímto by se detekovaly anomálie chodu systému, avšak klasifikátor musí mít apriorní znalost správných a chybových stavů. Tento přístup selže v případě profilace, která se výrazně

dynamicky mění v závislosti na příchozích požadavcích do sledovaného systému, tedy aktuálním stavu. Vhodnými metodami pro vyhodnocení správného chování programu jsou metody strojového učení či umělé inteligence, např. neuronové sítě, čímž na druhé straně stoupá složitost monitorovacího subsystému a jeho certifikace.

Z modulu rozhraní je tedy nutno získat minimálně následující informace o systému:

- Informace vykonávání daného úseku programu (např. z registru *Program Counter*),
- informaci o přerušení běhu úseku programu.

Model včasného vykonání vláken programu

Tento přístup je založen na kontrole, zda jednotlivá vlákna uživatelského programu dokáží svůj program vykonat do jejich definovaných deadlinů. Model tedy uchovává intervaly času vykonání vláken. Kontrolní modul by porovnával aktuální časy a potřebné časy vykonání dalších naplánovaných vláken s jejich deadliny. Nad tímto modelem a aktuálními daty by algoritmus (např. metodou Earlist Deadline First) spočítal aktuální pravděpodobnost vypršení deadlinů jednotlivých vláken. Výstupem by byla informace o chybě zmeškání deadline, ale i informace o konkrétním vláknu, které může nadřazený bezpečnostní systém ukončit, aby dal šanci ostatním vláknům. Tento přístup tedy dává nástroj pro řízenou degradaci systému. Tato technika by měla reagovat na defekty v systému ve srovnání s klasickým watchdogem dříve.

Z modulu rozhraní je tedy nutno získat minimálně následující informace o systému:

- Start běhu daného vlákna,
- přepnutí vlákna,
- aktuální informace běhu daného vlákna (např. z registru Program Counter).

8.3 Specifikace modelu

Formální model pro navrhovaný kontrolní modul vychází obecně z obecného orientovaného grafu a může mít dvě formy. První je rozšířená verze **časové Petriho sítě** a druhá je rozšířená verze **označeného transitního systému**.

Časové Petriho sítě je možné definovat dvěma způsoby. První možnost je definovat čas jako globální proměnnou a časové podmínky by tedy porovnávaly své hodnoty s touto proměnnou. Tato metoda je lepší pro implementaci v imperativním programovacím prostředí. Naopak pro prostředí s důrazem na paralelismus je lepší druhá možnost, a to definovat čas jako atribut markeru v barevné Petriho síti. Tímto je tedy nutné, aby všechny aktivní markery tento atribut pravidelně obnovovaly.

Rozšíření či funkcionalita, kterou je nutno obohatit oba uvažované modelovací nástroje, jsou monitorování, kontrola správnosti přechodů a kontrola časových podmínek.

8.3.1 Monitorování

Funkcionalita monitorování přisuzuje modelu možnost sledovat chod daného systému/programu. Síť tedy musí měnit svoje značení/stav v závislosti na příchozích monitorovaných informacích.

V případě **časové Petriho sítě**, resp. **barevné Petriho sítě** je tento atribut možné zakomponovat ve formě tzv. *hraničních podmínek*, což jsou podmínky, při jejichž splnění je daný marker (barva) zprocesován přechodem do dalšího místa. Další možností je definovat nový atribut sítě, který bude monitorované informace (např. události nebo hodnotu registru Program Counter) porovnávat se sítí.

V případě **označeného časového transitního systému** může být použito *označení* jakožto vstupů do modelu, a to monitorovaných informací typu *událost* i *hodnota registru Program Counter*, ale pouze v případě plné specifikace modelu. Model tedy musí zahrnovat případy, kdy přerušovací rutina může přerušit chod programu v kterémkoliv stavu.

8.3.2 Kontrola správnosti přechodů

Kontrola správnosti přechodů je základní funkcionalitou celého navrhovaného systému, díky němuž je možné chyby detekovat. Při analýze možností implementace této funkcionality vyvstaly dvě otázky. První otázkou je, zda tuto funkcionalitu zakomponovat do modelu, nebo do vyhodnocovacího procesu operujícího nad modelem. Druhou otázkou je, jakým způsobem tuto funkcionalitu vlastně implementovat.

V případě implementace výše zmíněné funkcionality do nadřazeného procesu bude muset takový proces existovat. Jeho čas vykonání musí být snížen na minimum z důvodu nepředvídatelné nejnižší časové vzdáleností dvou událostí, která je však reálně omezena časem reakce na přerušení programu. Nadřazený proces by musel prohledávat model a hledat aktuální stav. V tomto případě by jeho výpočetní složitost byla lineární, tedy $\Theta = log(n)$.

V případě implementace této funkcionality do modelu je teoreticky možné dosáhnout pomocí některých implementací (např. VHDL) rychlosti vykonání až na limity propagace signálu. Závisí tedy na konkrétní implementaci, přičemž teoretický základ nevylučuje ani možnost paralelismu.

Po volbě **implementace kontrolní funkcionality přímo do modelu** se u modelu typu Petriho síť nabízí integrovat tuto kontrolu jako další možné přechody z jed-

notlivých míst, a to do chybových míst, tedy tuto kontrolu distribuovat do celého modelu. Tato možnost však vyžaduje úplnou definici podmínek těchto přechodů a detekční vlastnosti modelu jsou na těchto podmínkách silně závislé.

U transitního systému je situace obdobná, avšak z teorie vyplývá, že transitní systém se může nacházet právě v jednom stavu, což je problém pro distribuovanou kontrolu, jelikož tento nástroj neumožňuje teoreticky kontrolovat současně ostatní stavy.

8.3.3 Kontrola časových podmínek

Kontrola časových podmínek detekuje chyby v nedodržování definovaných časových intervalů úseků programu. Způsob integrace jsem zvolil obdobně jako způsob integrace kontroly správnosti přechodů. Rozdíl je v tom, že tato kontrola pracuje s časem jakožto atributem modelu. Tento atribut tedy musí být přístupný z každého místa modelu. Implementace kontroly do formálního modelu Petriho sítě je možná pomocí *hraničních podmínek*. Obdobně u transitního systému lze tuto kontrolu implementovat pomocí *označení*.

8.3.4 Volba modelovacího nástroje

Volba modelovacího nástroje není jednoznačná. I když jsou Petriho sítě určeny pro modelování distribuovaných systémů a transitní sítě jsou určeny pro matematický popis diskrétních systémů, resp. programů, často se v literatuře modelují programy obzvláště v *konkurentním* prostředí operačního systému právě pomocí Petriho sítí. Pro také hovoří množství studií a nástrojů, které umožňují takovou Petriho síť vyšetřit z hlediska obsahu *deadlocku*. Dosažitelnost rozšířených *časových Petriho sítí* o další atributy je avšak obtížné až nemožné matematicky úplně vyšetřit [139].

V literatuře (např. [9]) se uvádí, že v *model checking* technice lze použít jako testovací model programu tzv. *Kripkeho struktury*, které tvoří *označený transitní systém* nad specifickou *výrokovou proměnnou*, která je tvořena *atomickou formulí*. Nevýhoda tohoto přístupu spočívá v exponenciálně rostoucí komplexitě (viz lemma 8.3.1).

Lemma 8.3.1. *Komplexita Kripkeho struktur s nárůstem testovaných proměnných systému (programu) je* $\Theta = log(x^n)$*. Naproti tomu komplexita Petriho sítě bude při stejném nárůstu* $\Theta = log(n)$*, protože se budou pouze přidávat místa, resp. hrany.*

Proti použití Petriho sítě hovoří fakt, že z formální definice operační sémantiky přímo vyplývá, že přechod mezi *místy* je nedeterministický, tedy čas odpalu přechodu (angl. *token firing*) není od okamžiku povolení (angl. *token enabled*) ohraničen. V [83] autoři uvádí proces, jak je možné pomocí změny operační sémantiky tento nedostatek

potlačit, resp. odstranit. Za určitých podmínek pak lze síť provozovat, avšak nelze modelovat efekty plynoucí z konkurentního běhu.

Další zvažované hledisko je způsob zaznamenání aktuální pozice programu v modelu. V případě *označeného transitního systému* by aktivní stav kopíroval aktuální pozici v programu, protože stav transitního systému je úzce spjat s aktuálním stavem (viz lemma 8.3.2), jelikož modeluje tento systém jako *deterministický* nikoliv *konkurentní*.

Lemma 8.3.2. *Stav označovaného transitního systému modelujícího program je v daném čase ekvivalentní s aktuální pozicí programového čítače, tedy* $S(\tau_i) \iff \Gamma(PC)$.

Z tohoto důvodu je tento modelovací nástroj vhodný kromě dalších případů (např. modelování PLC programu) pro modely založené na monitorování přesného stavu systému (tedy informaci z registru *Program Counter*), jako je např. *model včasného vykonání vláken programu*, *model profilace* nebo *model cest programu* (angl. program flow graph).

V případě *Petriho sítě*, která je určena pro modelování konkurentních systémů, může být *značení* přítomno i v *místě*, které neodpovídá aktuální pozici programu, ale pozici, na kterou může procesor skočit. Tato výhoda se uplatní při použití modelů založených na monitorování událostí systému.

Výsledkem tedy je volba modelovacího nástroje Petriho sítě pro vytvoření modelu událostí programu, resp. operačního systému reálného času z výše zmíněných důvodů i za cenu, že některé aspekty (hlavně *token firing*) definice Petriho sítě zavádí do modelu nedeterminismus. Tento nedostatek bude odstraněn pomocí změny definice operační sémantiky.

8.3.5 Formální definice

Modelovací nástroj pro vytvoření konkrétního modelu událostí operačního systému reálného času tedy formálně definuji jak z hlediska notace, tak z hlediska operační sémantiky.

Definice 1: Nechť je dána **kontrolní Petriho síť** rozšiřující standardní (časovou – barevnou) Petriho síť jako $RCPN = (P, T, pre, post, M_0, I, \Gamma, F, \phi)$, kde:

- P je konečná neprázdná množina míst;
- T je konečná neprázdná množina přechodů;
- $\bullet e : P \times T \rightarrow \mathbb{N}$ je zpětná přechodová funkce;
- $e\bullet : P \times T \rightarrow \mathbb{N}$ je dopředná přechodová funkce;
- $e \subseteq (P \times T) \cup (T \times P)$ je relace toku sítě reprezentovaná hranami;
- M_0 je počáteční značení;

- $I : T \to \mathbb{R}^+ \times (\mathbb{R}^+ \cup \infty)$ je funkce přiřazující přechodu nejmenší a nejvyšší hodnotu času;
- $\Gamma : P \to (\mathbb{N}^+)^2$ je funkce mapující počáteční a koncovou adresu úseku programu k místu, tedy $\Gamma \in [\Gamma_\alpha(t), \Gamma_\beta(t)]$;
- $\Psi : T \to \mathbb{N}^+$ je funkce nastavující adresu přechodu na následující úsek programu;
- F je konečná neprázdná množina chybových stavů;
- $\phi \subseteq (F \times T) \cup (T \times F)$ je relace hran a chybových stavů;

Množina F je také množina míst jako množina P, takže by hypoteticky mohla být její podmnožinou. Avšak místa v množině P zastupují model monitorovaného subsystému a místa v množině zastupují chybové stavy. Místa v množině F tedy už neobsahují atributy adres programu jako místa v množině P. Záměrem je napojit tyto chybové stavy do modulu výkonu bezpečnostních funkcí (angl. *Fault Collection and Control Unit*), chovají se tedy jako výstupy sítě.

Definice 2: Nechť jsou dány značení dle notační sémantiky Petriho sítí:

- $M : P \to \mathbb{N}$ je funkce značení;
- $pre(t)$ a $post(t)$ jsou vektory vyplývající z přechodové funkce hrany t, jedná se tedy o vektory $\bullet e(t_i)$ a $e \bullet (t_i)$;
- $I(t)$ je definováno jako $[I_\alpha(t), I_\beta(t)]$ jakožto nejmenší a nejvyšší možná hodnota odpalu hrany t;
- $act(M, p)$ označuje aktivní značení (marker) M a místo p;
- $\nu \in (\mathbb{R}^+)^n$ reprezentuje atribut časové značky ν_i markeru v případě, že je aktivní, tedy $act(M) \geq \bullet e(t_i)$;
- přechod t_i je odpálen v případě, kdy místo obsahuje aktivní marker $act(M) \geq \bullet e(t_i)$, když časový atribut markeru leží v mezích přechodu $\nu_i \in [I_\alpha(t_i), I_\beta(t_i)]$ a když hodnota registru *Program Counter* sledovaného subsystému dosáhla adresy pro přechod na další úsek programu (splněna hraniční podmínka následujícího přechodu), tedy $PC = \Psi(t_k)$;
- p_i místo je aktivní právě když $\uparrow enabled(t_k, M, t_i) = ((t_i = t_k) \vee (M - \bullet e(t_i) < \bullet e(t_k))) \wedge (M - \bullet e(t_i) + e \bullet (t_k))$
- odpálením přechodu vznikne nové značení $M' = M - \bullet e(t_i) + e \bullet (t_i)$.

Operační sémantika Petriho sítě může být postavena nejvýhodněji na definici pro **časový přechodový systém** (angl. *timed transition system*), který v sobě zahrnuje definici **označeného přechodového systému** obohaceného o atribut časové značky (viz [32]). Tento systém tedy poskytuje diskrétní složku pro události systému a spojitou složku pro vyhodnocení časového atributu barevné Petriho sítě.

Definice 3: Nechť je dána operační sémantika $RCPN$ jako časový přechodový systém $TTS = (S, S_0, A, C, I, \to)$, kde:

- S je množina stavů, přičemž počáteční stav $S_0 \subseteq S$;

- A je množina akcí;
- C je konečná množina proměnných obsahujících časovou značku;
- $I : S \to \Phi(C)$ přiřazuje časovou značku stavu;
- $\to \in S \times T \times 2^C \times S$ je binární relace nad množinou stavů, označovaná jako přechod.

Definice 4: Nechť jsou dány relace diskrétních a spojitých složek:

$$(M, \nu) \xrightarrow{t_i} (M', \nu') iff \begin{cases} PC = \Psi(t_i) \wedge \nu_i \in I(t_i) \\ M = pre(t_i) \end{cases} \tag{8.1}$$

$$(M, \nu) \xrightarrow{e_d} (M, \nu') iff \begin{cases} \nu_i' = \begin{cases} \nu_i + d & \text{if} PC \in \Gamma(p_i) \\ \nu_i' & \text{jinak.} \end{cases} \\ M \geq pre(t_i) \end{cases} \tag{8.2}$$

$$F' = F + \phi(t_i) iff \begin{cases} PC = \Psi(t_i) \wedge \nu_i \notin I(t_i) \\ PC \in \Gamma(p_i) \wedge M \neq pre(t_i) \\ (M, \nu) \xrightarrow{t_i} (M', \nu') \wedge M \geq pre(t_i) \end{cases} \tag{8.3}$$

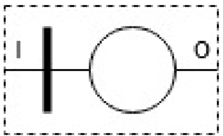
První rovnice popisuje diskrétní složku, tedy přechod značení do nového stavu po splnění příslušných podmínek, tj. register *Program Counter* dosáhl dané hodnoty a časová značka je v daných mezích. Druhá rovnice popisuje spojitou složku, tedy občerstvení časové značky barvy (markeru) za podmínky, že hodnota registru *Program Counter* je v mezích přiřazených k danému místu a místo obsahuje aktivní marker. Třetí rovnice popisuje nastavení chybových míst v případech, kdy jsou porušeny výše uvedené podmínky času a propagace markerů Petriho sítě.

8.3.6 Implementace kontrolního modulu

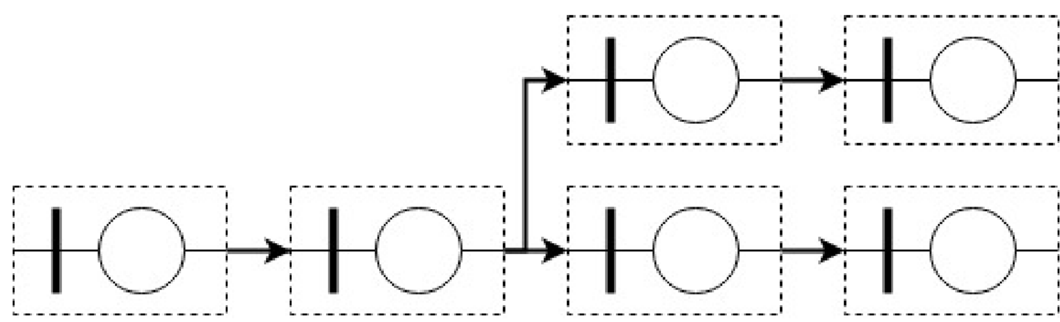
Implementace kontrolního subsystému se dělí z hlediska technických výzev na dvě části. První sadu technických problémů je potřeba vyřešit při integraci do systému, tedy rozhraní mezi kontrolním modulem a monitorovaným subsystémem. Možnosti technických řešení jsou diskutovány v kap. 5.1. Druhá sada problémů vyplývá z implementace Petriho sítě a její operační sémantiky do prostředí FPGA. Z kap. 4.7 je patrná schůdnost řešení implementace Petriho míst a hran jako funkčních objektů, které rozebírá i např. [109].

Při implementaci Petriho uzlu (místo a hrana) jsem vycházel z předpokladu, že vstupem hrany bude pouze jedno místo, a tím jsem mohl toto seskupení implementovat jako jeden uzel (zachyceno na obr. 8.3). Tyto uzly je možné seskupovat za účelem vytvoření programových linií a také větvit (viz příklad sítě na obr. 8.4).

Při vytváření VHDL kódu jsem postupoval dle matematického vyjádření (viz kap. 8.3.5). Jelikož jazyk VHDL neposkytuje matematické funkcionální paradigma, musel jsem jednotlivé funkce operační sémantiky sloučit a rozlišit pomocí logických výrazů. Výsledkem je kód uvedený v A.1.



Obr. 8.3: Diagram uzlu



Obr. 8.4: Příklad kontrolní Petriho sítě skládající se z uzlů

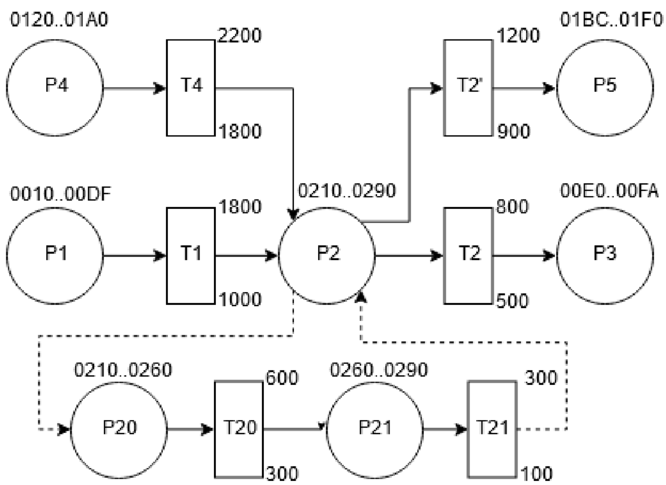
8.3.7 Integrace do vývoje cílové aplikace

Prvník krokem k vytvoření kontrolního subsystému, resp. kontrolního modulu je v ideálním případě vytvoření modelu monitorovaného systému dle zadané specifikace. Tento model by se měl ideálně sestávat z propojených návrhových vzorů (viz kap. 6.6) vyjádřených pomocí kontrolní Petriho sítě (viz ukázka implementace návrhového vzoru na obr. 8.5). Právě použití návrhových vzorů ve fázi návrhu patří mezi techniky minimalizující počet chyb. Tato metoda tedy předpokládá vytvoření knihovny návrhových vzorů a k nim příslušných modelů vyjádřených pomocí Petriho sítě.

8.4 Ukázka implementace návrhového vzoru synchronizačního objektu mutex

Implementace kontrolní Petriho sítě do VHDL kódu může být potom provedena:

- manuálním vytvořením propojených uzlů dle předpisu modelu,



Obr. 8.5: Ukázka implementace návrhového vzoru synchronizačního objektu mutex

- generováním VHDL kódu s následným doplněním adres z vygenerovaného mapovacího souboru vzniklého při kompilaci aplikace.

Ve fázi testování a verifikace systému se vytvořený kontrolní modul obsahující model se znalostí adres funkcí také uplatní při SIL, a dokonce HIL simulaci. Kontrolní subsystém se také může verifikovat pomocí metody injekce chyb (angl. *fault injection*).

8.5 Definice vzoru architektury

V této podkapitole se pokusím o definici vzoru architektury, jehož komponenty jsem doposud rozebíral. Pro vytvoření textové formy použiji strukturu, terminologii a doporučení dle [141]. Za grafickou formu definice může být považován diagram na obr. 8.1. Popis konkrétní implementace je zaznamenán v kap. 8.2. Jak už bylo řečeno v kap. 6.5.7, jedná se o vzor architektury, protože pracuje se software i hardware, ale může být označen i jako vzor návrhu. Nemám za cíl definovat finální verzi obecně platného vzoru, protože definice vzoru je iterativní proces a jeho definice musí být přijatelná relevantní skupinou lidí.

Název (angl. *name*): Online kontrolní systém procesorové jednotky.

Kontext (angl. *context*): Jste architekti nebo vývojáři systému odolnému proti poruchám, tedy se zvýšenými požadavky na funkční bezpečnost. Máte možnost napojit se na vnitřní sběrnici procesorové jednotky. Máte možnost použít SoC integrovaný obvod obsahující hradlové pole. Máte zájem o vývoj software dle metodiky modelem řízená architektura.

Problém (angl. *problem*): Potřebujete detekovat chyby RTOS (porušení časových podmínek, přeskočení programu, *deadlock*, *livelock*, *race condition* a deadline miss) a defekty, které se těmito chybami projeví (např. SEU, zkrat pinu s navázaným

přerušením, trvalé rozpojení transistoru nebo problém s napájením). Dále to mohou být chyby vyplývající z implementace uživatelské aplikace. Potřebujete také, aby tato detekce minimálně ovlivnila daný systém a spotřebovala minimální množství výpočetních zdrojů.

Aspekty (angl. *forces*): Chcete zlepšit diagnostické pokrytí systému, ale použité techniky nedetekují všechny potenciální chyby. Chcete integrovat kontrolní systém, ale chcete, aby jeho integrace nepřispívala k dalším potenciálním chybám. Chcete integrovat kontrolní systém, ale máte omezené možnosti připojení. Chcete integrovat další bezpečnostní funkce do systému, ale přitom neomezovat výpočetní výkon.

Řešení (angl. *solution*): Integrovat mechanismus pro procesorovou jednotku do propojeného hradlového pole, který bude tuto jednotku v reálném čase kontrolovat dle ověřeného modelu implementované aplikace za účelem detekce potenciálních chyb vznikajících v důsledku hardwarových nebo softwarových defektů.

Důsledky (angl. *consequences*): Mezi přínosy patří, že tato technika eliminuje chyby se společnou příčinou. Přínosem také je minimální zatížení monitorované procesorové jednotky v závislosti na stupni integrace (napojení na sběrnici procesoru nebo jako mapovaná periferie). Nevýhodou je, že musíte vytvořit model aplikace a ten formálně verifikovat. Zvýšíte ale diagnostické pokrytí, a tím pravděpodobně zlepšíte úroveň bezpečnosti systému.

8.6 Zhodnocení

Tato kapitola přináší popis komponent online kontrolního systému, popis jeho implementace a integrace do systému. Ústřední výkonnou částí tohoto systému je kontrolní modul, který zapouzdřuje model aplikace vytvořený pomocí navrhnuté kontolní Petriho sítě. Ta je definována pomocí notační a operační sémantiky. Kromě rozebraného modelu událostí RTOS přináší tato kapitola také návrh jiných typů modelů software vybavení, např. model včasného vykonání vláken a model profilace funkcí. Mapováním procesu použití tohoto nástroje do cyklu vývoje software vznikne komplexně specifikovaná automatizovatelná technika, kterou je možné použít jako řešení na podobnou sadu problémů, resp. jako vzor architektury.

Navrhnutá integrace kontrolního systému a způsob jeho tvorby koreluje s progresivní metodikou modelem řízená architektura. Vlastní implementace závisí na použité platformě. Napojení modulu rozhraní tvoří klíčový aspekt pro definici požadavku na minimální zatížení monitorovaného procesu a pro zaručení schopnosti detekce jeho pozastavené činnosti. Jelikož lze kontrolní subsystém sestavit z různých implementací modulu rozhraní a kontrolního modulu, jedná se o obecné řešení daného problému, tedy o návrhový vzor, resp. o vzor architektury.

9 SIMULACE

Tato kapitola se zabývá implementací kontrolního subsystému, resp. jeho simulací. Nejdříve definuji zvolené prostředí a nástroje (viz kap. 9.1). Potom představím simulovaný systém obsahující implementaci kontrolního systému jako celku (viz kap. 9.2), resp. různé architektury systému simulace, které jsem zkoušel implementovat. Po úspěšné implementaci systému simulace jsem zaznamenal a namodeloval záznam běhu reálné aplikace na reálném hardware, který bude sloužit jako vstup simulace (viz kap. 9.3). Dále jsem implementoval model kontrolovaného programu (viz kap. 9.4). Provedením simulace se záznamem neobsahujícím chybu a vnesením různých druhů chyb získám výsledky (viz kap. 9.5), které by měly potvrdit správnost implementace a detekce chyby pomocí kontrolního modulu, tedy jeho reakce na defekt v monitorovaných datech. V kap. 9.6 nakonec vyhodnotím výsledky provedené simulace.

9.1 Prostředí a nástroje

Jelikož je reálná implementace a integrace navrhována pro prostředí hradlového pole, implementoval jsem proto kontrolní subsystém pomocí nástroje **Quartus** a **ModelSim**, což jsou nástroje pro programování a simulaci hradlových polí firmy **Altera**. Jako programovací jazyk pro implementaci kontrolního modulu jsem zvolil **Verilog**, pro *node* element kontrolního modulu jsem použil VHDL a pro sestavení simulace (tzv. *testbench*) jsem zvolil **SystemVerilog**. Výsledkem simulace v programu ModelSim je graf časových průběhů jednotlivých signálů, ze kterého je možné vyčíst chování simulovaného modulu.

9.2 Architektura simulace

Při vytváření architektury simulace jsem kladl důraz na co nejmenší aproximaci simulovaného systému od reálného (plánovaného). Na začátku byla idea totálně reálné integrace, a to implementace kontrolního modulu jako periferní jednotky k CPU. Za účelem vytvoření proof-of-concept se jeví jako jediná schůdná cesta hardwarová implementace do hradlového pole s napojením na sběrnici CPU, přičemž jako reálné hardware mohu použít SoC, které sdružuje CPU a FPGA na jednom čipu, nebo implementovat CPU jako *soft-core* přímo do FPGA.

Jako hardware jsem zvolil vývojový kit **DE0-Nano-Soc** od firmy **Altera**, který obsahuje **Cyclone V** SoC. Důvodem volby Altera vs. Xilinx je existující port zvoleného RTOS na *soft-core* **Nios II**, protože MicroBlaze není podporován systémem

embOS. Tato skutečnost skýtá možnost napojení se přímo na vnitřní registry CPU, jako je **Program Counter**, což se ale ukázalo jako chybná úvaha, protože *soft-core* je implementován jako IP *core* a ve volné neplacené verzi neposkytuje možnost vnitřní modifikace (po bližším průzkumu jsem zjistil, že MicroBlaze tuhle možnost neposkytuje ani v placené verzi). Napojení na vniřní registry HPS procesoru taky není možné, protože propojení mezi HPS a FPGA částí je zajištěno pomocí tzv. *HPS-to-FPGA bridge*, který propojuje FPGA se sběrnicí ARM procesoru APB a AXI. Architektura založená na monitorování registru **Program Counter** tedy není v mém případě možná.

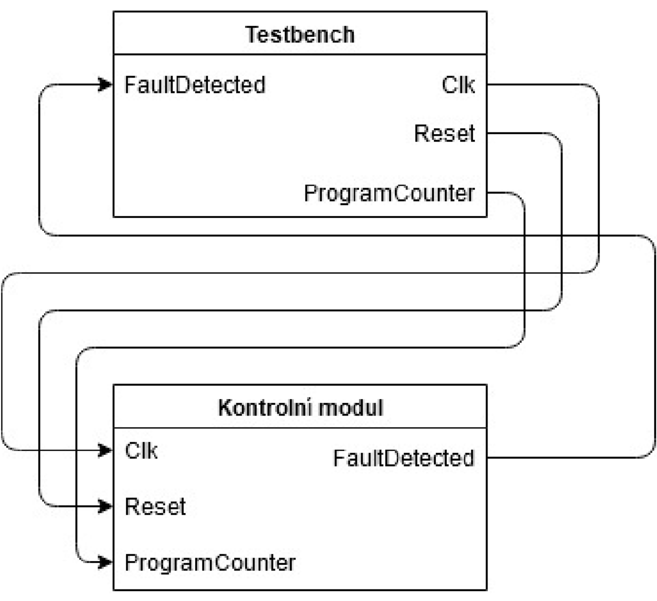
Jako operační systém jsem vybral **embOS**. Důvodem je bezplatná licence pro nekomerční účely, ale hlavně možnost integrace knihoven **Segger**, které spolu se software **SystemView** dokáží v reálném čase monitorovat běh tohoto RTOS, přenášet data o vykonaných funkcích pomocí zařízení **J-Link** do PC a zobrazovat je. Nejedná se o neinvazivní monitorování, protože tyto knihovny musí běžet na cílové platformě. Neinvazivní monitorování běhu systému by zajistilo zařízení **J-Trace**, které bohužel není k dispozici.

Nabízí se tedy možnost sestavit testovací architekturu založenou na monitorování reálného běhu RTOS pomocí **Segger** knihoven a vyhodnocovat detekci chyby kontrolním subsystémem. Při implementaci celého systému do SoC jsem ale narazil na technický problém, kdy **Segger** knihovny musí posílat monitorovaná data pouze pomocí **J-Link** z důvodu rychlosti přenášených dat (doporučována je komunikace na frekvenci alespoň 1 MHz). Problémem je, že vývojový kit **DE0-Nano-Soc** zapouzdřuje JTAG rozhraní debugovacím subsystémem **USB Blaster II**, takže není možné se k FPGA ani HPS připojit pomocí standardního rozhraní JTAG, které vyžaduje zařízení **J-Link**. Implementaci **J-Link** protokolu manuálně značně omezuje chybějící dokumentace. Tímto je pro mě znemožněno integrovat testovací architekturu založenou na monitorování běhu systému a monitorování správné indikace chyby kontrolního modulu.

Realizovatelnou testovací architekturou tedy zůstává implementace aplikace a **embOS** na **Nios**, implementace kontrolního modulu do FPGA a jako spojení použít rozhraní mapované periferie (pomocí sběrnice Avalon), přičemž budou do kontrolního subsystému posílány vzniklé události v reálném čase (zápisem na sběrnici). Kvůli omezeným možnostem debugování tento způsob technické realizace předpokládá integraci již odzkoušených subsystémů. Další možností je tuto testovací architekturu nejdříve simulovat pomocí nástroje **Modelsim**, kde je možné monitorovat všechny signály systému, a tedy lépe ladit. Problémem ale je omezená licence simulačního prostředí na určitý počet řádků kódu (na 10 000 řádků), do čehož jsem se s 13 270 řádky kódu testovacího systému nevešel. Řešením tedy je simulovat nejdříve jednotlivé subsystémy zvlášť.

Rozhodl jsem se tedy nejdříve simulovat samotný kontrolní subsystém. Rozhraní tohoto modulu tvoří sběrnice **Avalon**, pomocí které modul dostává informace o událostech RTOS. Abych mohl v této podobě modul odzkoušet, musel bych vytvořit komplexní *testbench*, který disponuje subsystémy pro řízení této sběrnice (mód *master*).

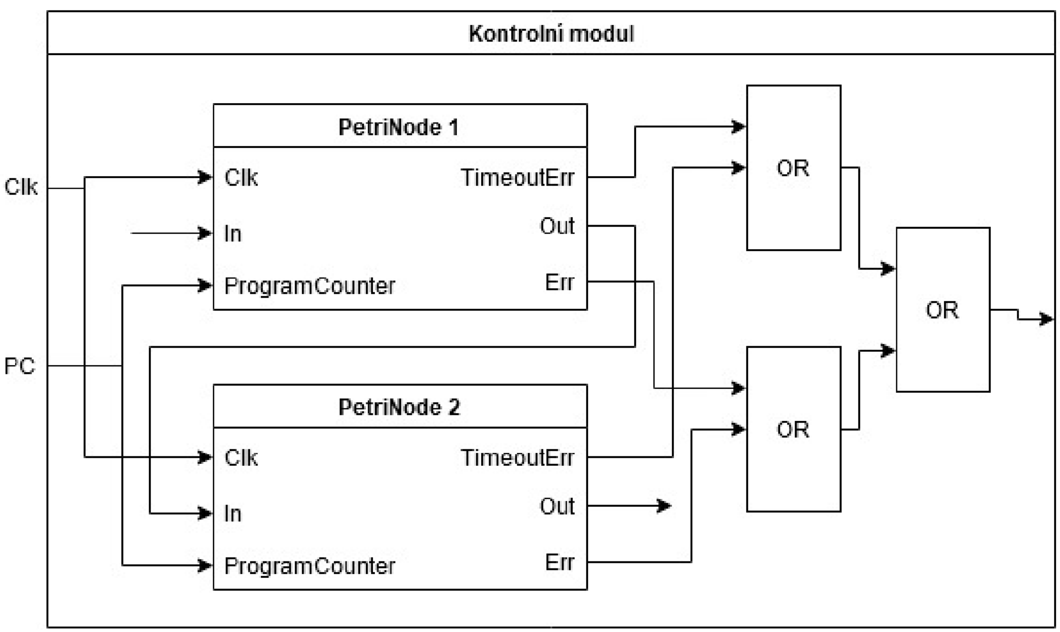
Jelikož kontrolní subsystém pak obsahuje rozhraní pro tuto sběrnici v módu *slave*, rozhodl jsem se testovací architekturu ještě více zjednodušit, a to nadefinováním rozhraní, které přenáší přesně ta data, která kontrolní modul potřebuje. Díky tomu mohu také použít model zahrnující hodnotu **Program Counter**. Blokové schéma testovací architektury zachycuje obrázek 9.1. Blokové schéma samotného kontrolního modulu (viz obr. 9.2) ukazuje, že základním vyhodnocovacím modulem je *PetriNode*, který implementuje místo a přechod jakožto jeden uzel **kontrolní Petriho sítě** (viz kap. 8.3.6). Definici modulu *PetriNode* zachycuje obr. 9.3.



Obr. 9.1: Blokové schéma testovací architektury

9.3 Testovací scénář

Pro simulaci jsem vybral z definovaných testovacích scénářů (viz kap. 7.3) scénář „Deadlock“. Jelikož ale simuluji a testuji pouze vlastní kontrolní modul, musím namodelovat vstup do tohoto modulu. Proto jsem část záznamu událostí (viz obr. 9.4) získaný z měření reálně běžícího systému provedeného v kap. 7 převedl na sadu příkazů definující časově se měnící signál (hodnota *Program Counter*) pro vytvořený *testbench*. Vybral jsem pro ověření simulace pouze část celého záznamu událostí



Obr. 9.2: Blokové schéma kontrolního modulu

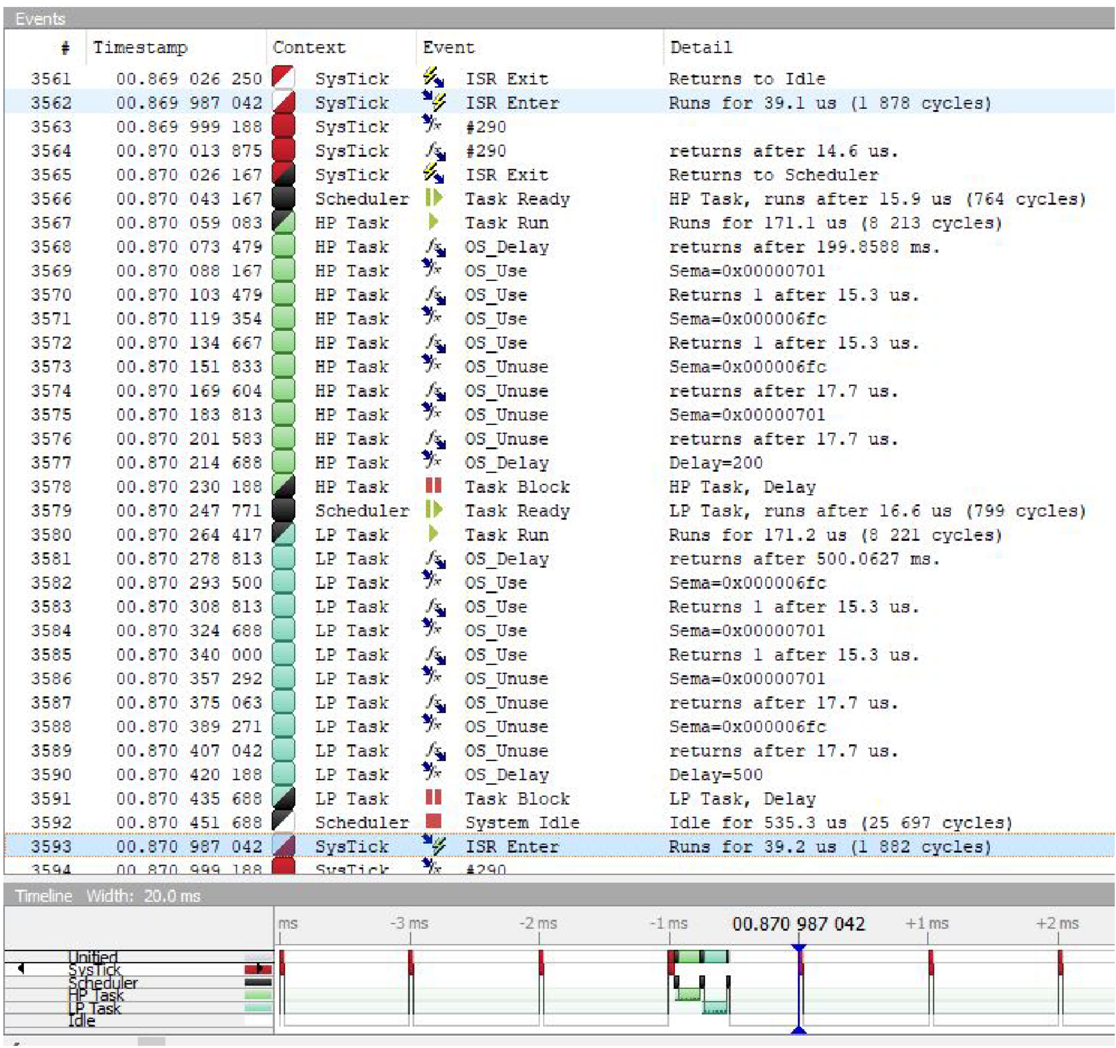
```
entity petri_node is
    generic (
        pc_start : natural := 5;
        pc_end : natural := 10;
        earliest : natural := 10;
        latest : natural := 20
    );
    port (
        clk: in std_logic;
        pc: in natural;
        in_transition: in std_logic;
        out_transition: out std_logic;
        timeout: out std_logic;
        marking: out std_logic;
        err: out std_logic
    );
end petri_node;
```

Obr. 9.3: Definice modulu *PetriNode* v jazyce VHDL

systému, a to tu část, kde se nejvíce projeví přepínání vláken obsluhy přerušení, hlavního a vedlejšího.

Převedený signál potom slouží jako stimul pro simulovaný modul. Časový výpis tohoto stimulu zachycuje obrázek 9.5. V prvním sloupci tohoto výpisu se nachází čas výskytu události popsané ve druhém sloupci. Při definování časových intervalů mezi událostmi jsem změnil časové měřítko. Pro simulaci jsem použil časové rozlišení 1 ns, abych smysluplně zobrazil změny jednotlivých signálů na hardwarové úrovni, přičemž

jsem uvažoval o nasazení do mikroprocesorových systémů s taktovacími frekvencemi v řádech 10–100 MHz. Aby kromě zobrazení simulace byla i délka simulace v řádech sekund, změnil jsem časové měřítko z původních mikrosekund na nanosekundy.



Events

#	Timestamp	Context	Event	Detail
3561	00.869 026 250	SysTick	ISR Exit	Returns to Idle
3562	00.869 987 042	SysTick	ISR Enter	Runs for 39.1 us (1 878 cycles)
3563	00.869 999 188	SysTick	#290	
3564	00.870 013 875	SysTick	#290	returns after 14.6 us.
3565	00.870 026 167	SysTick	ISR Exit	Returns to Scheduler
3566	00.870 043 167	Scheduler	Task Ready	HP Task, runs after 15.9 us (764 cycles)
3567	00.870 059 083	HP Task	Task Run	Runs for 171.1 us (8 213 cycles)
3568	00.870 073 479	HP Task	OS_Delay	returns after 199.8588 ms.
3569	00.870 088 167	HP Task	OS_Use	Sema=0x00000701
3570	00.870 103 479	HP Task	OS_Use	Returns 1 after 15.3 us.
3571	00.870 119 354	HP Task	OS_Use	Sema=0x000006fc
3572	00.870 134 667	HP Task	OS_Use	Returns 1 after 15.3 us.
3573	00.870 151 833	HP Task	OS_Unuse	Sema=0x000006fc
3574	00.870 169 604	HP Task	OS_Unuse	returns after 17.7 us.
3575	00.870 183 813	HP Task	OS_Unuse	Sema=0x00000701
3576	00.870 201 583	HP Task	OS_Unuse	returns after 17.7 us.
3577	00.870 214 688	HP Task	OS_Delay	Delay=200
3578	00.870 230 188	HP Task	Task Block	HP Task, Delay
3579	00.870 247 771	Scheduler	Task Ready	LP Task, runs after 16.6 us (799 cycles)
3580	00.870 264 417	LP Task	Task Run	Runs for 171.2 us (8 221 cycles)
3581	00.870 278 813	LP Task	OS_Delay	returns after 500.0627 ms.
3582	00.870 293 500	LP Task	OS_Use	Sema=0x000006fc
3583	00.870 308 813	LP Task	OS_Use	Returns 1 after 15.3 us.
3584	00.870 324 688	LP Task	OS_Use	Sema=0x00000701
3585	00.870 340 000	LP Task	OS_Use	Returns 1 after 15.3 us.
3586	00.870 357 292	LP Task	OS_Unuse	Sema=0x00000701
3587	00.870 375 063	LP Task	OS_Unuse	returns after 17.7 us.
3588	00.870 389 271	LP Task	OS_Unuse	Sema=0x000006fc
3589	00.870 407 042	LP Task	OS_Unuse	returns after 17.7 us.
3590	00.870 420 188	LP Task	OS_Delay	Delay=500
3591	00.870 435 688	LP Task	Task Block	LP Task, Delay
3592	00.870 451 688	Scheduler	System Idle	Idle for 535.3 us (25 697 cycles)
3593	00.870 987 042	SysTick	ISR Enter	Runs for 39.2 us (1 882 cycles)
3594	00.870 999 188	SysTick	#290	

Obr. 9.4: Záznam událostí reálně běžícího scénáře

Vnitřní vybavení kontrolního modulu je odvozeno od taktovacího signálu, aby tento modul byl synchronizován s monitorovaným subsystém. Proto jsem ještě upravil časové měřítko, aby kontrolní modul mohl reagovat v čase na stimulus. Převodem 1 ns na 1hodinový cyklus systému jsem při hodinovém cyklu 10 *ns* (100 MHz) zvětšil měřítko stimulu 10x, přičemž tedy výsledný stimulus je časově oproti původnímu zmenšen 100krát. Díky změně časového měřítka jsem tedy simulaci zrychlil a také z toho vyplývá fakt, že pokud bude kontrolní modul časově zvládat zpracovávat všechny události, nebude mít pak problém zpracovávat události z reálného systému. Na obrázku 9.6 je úsek kódu pro *testbench* definující stimulus.

```
#                     0 Stimuli started
#                   987 Systick: ISR Enter
#                  1000 Systick: Start ISR routine #290
#                  1015 Systick: Finish ISR routine #290
#                  1028 Systick: ISR Exit
#                  1045 Scheduler: Task Ready
#                  1061 HP task: Task Run
#                  1075 HP task: Finish OS_Delay 200
#                  1090 HP task: Start OS_Use 0x0701
#                  1105 HP task: Finish OS_Use 0x0701
#                  1121 HP task: Start OS_Use 0x06fc
#                  1136 HP task: Finish OS_Use 0x06fc
#                  1153 HP task: Start OS_Unuse 0x06fc
#                  1171 HP task: Finish OS_Unuse 0x06fc
#                  1185 HP task: Start OS_Unuse 0x0701
#                  1203 HP task: Finish OS_Unuse 0x0701
#                  1216 HP task: Start OS_Delay 200
#                  1232 HP task: Task Block
#                  1249 Scheduler: Task Ready
#                  1266 LP task: Task Run
#                  1280 LP task: Finish OS_Delay 500
#                  1296 LP task: Start OS_Use 0x06fc
#                  1311 LP task: Finish OS_Use 0x06fc
#                  1327 LP task: Start OS_Use 0x0701
#                  1342 LP task: Finish OS_Use 0x0701
#                  1359 LP task: Start OS_Unuse 0x0701
#                  1377 LP task: Finish OS_Unuse 0x0701
#                  1391 LP task: Start OS_Unuse 0x06fc
#                  1409 LP task: Finish OS_Unuse 0x06fc
#                  1422 LP task: Start OS_Delay 500
#                  1437 LP task: Task Block
#                  1453 Scheduler: System Idle
#                  1987 Systick: ISR Enter
#                  2000 Systick: Start ISR routine #290
#                  2015 Systick: Finish ISR routine #290
#                  2028 Systick: ISR Exit
```

Obr. 9.5: Výpis časového signálu jako stimulu pro simulaci

9.4 Implementace modelu programu

Pro správnou funkci kontrolního modulu a detekci chyb v kontrolovaném vzorku vstupních dat jsem vytvořil model programu, ze kterého byl tento vzorek extrahován. Model programu obsahuje síť modulů *Petri node* spolu s definicí časových limitů a příslušných rozsahů adres *Program Counteru* pro jednotlivé uzly. Ukázka implementace takového modelu v jazyku *Verilog* obsahující program s vlákny a jejich vnitřními systémovými funkcemi je zachycena na obr. 9.7.

Jelikož se jedná o program běžící na RTOS, a systém tedy pravidelně vykonává systémové funkce nepatřící vláknům, modeloval jsem běh těchto funkcí do separátní *kontrolní Petriho sítě*. Modelováním jednotlivých vláken programu a jeho jádra vzniklo několik separátních sítí. Nabízí se spojit tyto jednotlivé sítě jako *hierarchickou Petriho síť*. V našem případě výseku záznamu událostí běhu programu by tato operace nevadila, avšak při reálném běhu mohou nastat problémy při implementaci tohoto teoretického úkonu, např. ošetřit případ, kdy je vlákno v kterémkoliv okamžiku

```
program_counter = 205;
$display($time,"\tHP task: Task Run");
repeat(13) @(posedge clk);//#14;
program_counter = 209;
repeat(1) @(posedge clk);

program_counter = 35;
$display($time,"\tHP task: Finish OS_Delay 200");
repeat(14) @(posedge clk);//#15;
program_counter = 39;
repeat(1) @(posedge clk);

program_counter = 40;
$display($time,"\tHP task: Start OS_Use 0x0701");
repeat(14) @(posedge clk);//#15;
program_counter = 44;
repeat(1) @(posedge clk);

program_counter = 45;
$display($time,"\tHP task: Finish OS_Use 0x0701");
repeat(15) @(posedge clk);//#16;
program_counter = 49;
repeat(1) @(posedge clk);

program_counter = 50;
$display($time,"\tHP task: Start OS_Use 0x06fc");
repeat(14) @(posedge clk);//#15;
program_counter = 54;
repeat(1) @(posedge clk);

program_counter = 55;
$display($time,"\tHP task: Finish OS_Use 0x06fc");
repeat(16) @(posedge clk);//#17;
program_counter = 59;
repeat(1) @(posedge clk);
```

Obr. 9.6: Ukázka definice stimulu pro *testbench* v jazyku SystemVerilog

```
// hptask
petri_node #(35, 39, 13, 17) node_hptask_osdelay_finish (.clk(clock), .pc(program_counter), .in_transition(sched_task_run), .out_transition(
petri_node #(40, 44, 13, 17) node_hptask_osuse_start1 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osdelay_finish), .out_tr
petri_node #(45, 49, 14, 18) node_hptask_osuse_finish1 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osuse_start1), .out_tra
petri_node #(50, 54, 13, 17) node_hptask_osuse_start2 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osuse_finish1), .out_tra
petri_node #(55, 59, 15, 19) node_hptask_osuse_finish2 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osuse_start2), .out_tra
petri_node #(60, 64, 16, 20) node_hptask_osunuse_start2 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osuse_finish2), .out_t
petri_node #(65, 69, 12, 16) node_hptask_osunuse_finish2 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osunuse_start2), .out
petri_node #(70, 74, 16, 20) node_hptask_osunuse_start1 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osunuse_finish2), .out
petri_node #(75, 79, 11, 15) node_hptask_osunuse_finish1 (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osunuse_start1), .out
petri_node #(80, 84, 14, 18) node_hptask_osdelay_start (.clk(clock), .pc(program_counter), .in_transition(app_hptask_osunuse_finish1), .out_

// lptask
petri_node #(95, 99, 14, 18) node_lptask_osdelay_finish (.clk(clock), .pc(program_counter), .in_transition(sched_task_run), .out_transition(
petri_node #(100, 104, 13, 17) node_lptask_osuse_start2 (.clk(clock), .pc(program_counter), .in_transition(app_lptask_osdelay_finish), .out_
petri_node #(105, 109, 14, 18) node_lptask_osuse_finish2 (.clk(clock), .pc(program_counter), .in_transition(app_lptask_osuse_start2), .out_t
petri_node #(110, 114, 13, 17) node_lptask_osuse_start1 (.clk(clock), .pc(program_counter), .in_transition(app_lptask_osuse_finish2), .out_t
```

Obr. 9.7: Ukázka definice modelu pomocí *Petri node* v jazyku *Verilog*

přerušeno.

Při modelování jádra RTOS, resp. plánovače úloh jsem tedy prošel zaznamenané typy sledu událostí systému a vytvořil separátní *kontrolní Petriho síť*. Kontinuitu

událostí jsem tedy vyřešil zajištěním volnosti přechodu do podružných *nodů*. Tento způsob implementace tedy znemožňuje kontrolovat správnost výběru správného vlákna nebo přerušení plánovačem. Také jsem nenamodeloval všechny možné stavy plánovače, ale pouze ty, které byly aktivní při měření. Tato implementace modelu plánovače je zachycena na obr. 9.8.

```
// scheduler
petri_node #(200, 204, 14, 18) node_sched_task_ready (.clk(clock), .pc(program_counter), .in_transition((sched_idle & ap
petri_node #(205, 209, 12, 16) node_sched_task_run (.clk(clock), .pc(program_counter), .in_transition(sched_task_ready),
petri_node #(210, 214, 15, 19) node_sched_task_block (.clk(clock), .pc(program_counter), .in_transition(sched_idle & (ap
petri_node #(215, 219, 14, 18) node_sched_task_idle (.clk(clock), .pc(program_counter), .in_transition(sched_task_block)
always @(posedge clock)
begin
    if (sched_task_run) begin
        sched_idle <= 1;
    end
    if (sched_task_idle) begin
        sched_idle <= 1;
    end
    if (sched_task_ready) begin
        sched_idle <= 0;
    end
    if (sched_task_block) begin
        sched_idle <= 0;
    end
end
```

Obr. 9.8: Modelování plánovače RTOS jako separátní *kontrolní Petriho síť*

9.5 Výsledky simulace

Provedením simulace v prostředí **ModelSim** jsem získal průběhy jednotlivých vnitřních signálů kontrolního modulu jako odezvy na definovaný stimul. Hlavním výstupem simulace ale je ověření celkové funkčnosti implementace a prvotní ověření správné detekce chyby v kontrolovaných datech, tedy indikace signálu *FaultDetected*. Tento signál reaguje na porušení časového limitu vykonávání jednotlivých uzlů Petriho sítě a také na neoprávněné vykonávání uzlu (případ, kdy je porušena kontinuita přechodů mezi uzly).

Simulaci jsem rozdělil na třídy, kdy jednotlivé třídy zastupují situace stejného typu běhu systému, tedy:

- třída I – normální běh,
- třída II – časová chyba,
- třída III – chyba kontinuity.

9.5.1 Třída I – normální běh

Jedná se o situaci, kdy není vnesena žádná chyba a stimul odpovídá běhu reálného programu. Dle očekávání tedy kontrolní modul na výstupu *FaultDetected* nesignalizuje

chybu. Tento stav tedy v reálných aplikací probíhá neustále. Časový záznam se nachází v příloze A.12.

9.5.2 Třída II – časová chyba

Tato třída zastupuje situace, kdy nastala chyba v důsledku porušení časových limitů vykonávání některé z událostí (funkcí systému). Je tedy vnesena chyba v podobě prodloužení času přechodu mezi událostmi. Tato situace odpovídá případu, kdy se vykonávání dané systémové funkce zpozdí nad definovanou mez, což v deterministickém systému není žádoucí. Kontrolní modul tedy pomocí výstupu *FaultDetected* signalizuje chybu. Časový záznam se nachází v příloze A.13.

Chyba byla vnesena prodloužením události s adresou $C8$ z 16 cyklů na 30 cyklů, což odpovídá 12hodinovým cyklům navíc nad limit této události. Kontrolní modul tedy detekoval chybu v čase 1hodinového cyklu od meze 18 cyklů na událost, tedy v čase 10 ns od výskytu chyby.

9.5.3 Třída III – chyba kontinuity

Jedná se o situace, kdy nastala chyba porušení pravidel kontinuity (sledu událostí) dle modelu. Je tedy vnesena chyba v podobě zápisu jiné hodnoty registru *Program Counter*, než v dané situaci může být. Tato situace odpovídá případu, kdy se vlivem defektu (nebo cíleného útoku) přepíše registr *Program Counter* a začne se vykonávat nežádoucí (jiný) úsek kódu. Kontrolní modul tedy nastaví svůj výstup *FaultDetected*, čímž signalizuje chybu. Časový záznam se nachází v příloze A.14.

Chyba byla vnesena přepsáním hodnoty *Program counter* z CD na 14 u události *task_run*, která následuje po události s adresou $C8$. Kontrolní modul reagoval v čase 1hodinového cyklu, tj. 10 ns od projevu této chyby.

9.6 Hodnocení výsledků

Dle výsledků simulace uvedených v kap. 9.5 mohu konstatovat, že kontrolní modul úspěšně detekoval chyby v testovací sadě. Abych mohl deklarovat plnou funkčnost, musí být ještě provedena verifikace, což je komplexní činnost přesahující rámec této práce. Z programátorského hlediska ale mohu prohlásit, že kontrolní modul složený z uzlů a vyhodnocovací logiky je schopný plnit svou funkci. Tedy aktivita jednotlivých uzlů přechází nepřerušeně v závislosti na hodnotě registru *Program Counter*.

Jelikož jsem omezil vstupní stimul pouze na úsek záznamu událostí systému, není vyloučeno, že se nevyskytnou situace, kdy bude odpověď kontrolního modulu klasifikována jako *falešný poplach* nebo *nedetekovaná chyba*. Tento problém je možné

vyřešit pouze specifikací všech typů situací s následnou verifikací *kontrolní Petriho sítě*.

Z výsledků také vyplývá fakt, že kontrolní modul musí obsahovat model programu s korektně definovanými časovými intervaly, rozsahy fyzických adres a všechny události (uzly) musí být správně pospojovány. U mé testovací aplikace se jednalo o jednoduchý příklad dvou vláken využívajících mutexy. Model tedy obsahoval několik uzlů, které přesně kopírovaly sled všech událostí v jednotlivých vláknech programu. Pro komplexní programy bude ale model několikanásobně větší (paměťová náročnost), tedy bude výhodné ho tvořit strojově. Další možností je modelovat program pouze z některých událostí (např. z těch, kde se program větví, nebo jsou stěžejní).

Implementace kontrolního modulu obsahuje z programátorského hlediska minimum registrů (paměťové místo v sekvenční logice). Uzel *kontrolní Petriho sítě* obsauje pouze dva registry, ale počet těchto modulů narůstá s komplexitou programu lineárně. Optimalizací by mohlo jít dosáhnout redukce na jeden registr, což si myslím, že je minimum pro zajištění správné funkce a implementace dle definice.

Kontrolní modul má dle vytyčených cílů detekovat chyby v běhu programu. Fakticky kontrolní modul kontroluje časovou správnost kontinuity běhu programu dle modelu. V tomto okamžiku tedy mohu nadnést logickou úvahu ohledně detekce *deadlock*, *livelock* a *starvation*. Tedy v případě, že model neobsahuje výše zmíněné negativní jevy (což je možné dokázat pomocí formální verifikace vlastností modelu), neexistuje povolená cesta v modelu kontrolního modulu, a tedy všechny ostatní cesty vykonávání programu (potenciálně obsahující defekty projevující se jako výše zmíněné jevy) jsou označeny jako chyba. Složitost takové formální verifikace modelu programu jsem ale již nastínil v kap. 4.7.

V porovnání s *watchdog* (nejběžnější standardní technika pro detekci anomálií v software) dokazuje simulace řádově rychlejší detekci. Přesný poměr závisí na střední hodnotě maximálních časových mezí uzlů (oproti reálným hodnotám) vztažené k *watchdog* oknu (hodnoty se standardně pohybují v rozmezí 0,5–2 s).

U obou metod existuje potenciál poruch typu CCF (např. celkový výpadek napájení a deviace hodinového signálu), avšak právě způsob hardwarové implementace kontrolního modulu a jeho napojení některým z těchto chyb zamezují. Bez důkladné analýzy vlivu CCF na vytvořený kontrolní modul je ale možné nějaké závěry pouze odhadovat.

10 ZÁVĚR

Z obecného cíle vyvinout podpůrný algoritmus pro zlepšení funkční bezpečnosti embedded systému obsahujícího RTOS, který má být navíc implementovatelný v hardware, jsem průzkumem současné literatury popsal několik podoblastí, ve kterých existují nevyřešené problémy, výzvy nebo existuje prostor pro další hlubší výzkum. Z průzkumu vědecké literatury v těchto oblastech vyplývá, že některé problémy lze řešit mnoha rozličnými přístupy, avšak do některých oblastí se pouští málo vědeckých osobností. Zaměřil jsem se proto v této práci na oblast verifikace běhu embedded programu obsahujícího RTOS za jeho běhu.

Zaměřil jsem se tedy na detekci chyb, což je první fáze procesu systémů odolných poruchám. Vytyčením oboru detekovaných chyb na porušení podmínek chodu programu, což se může projevovat i jako *deadlock*, *livelock* a *starvation*, jsem se rozhodl pro metodu kontroly běhu systému dle modelu, přičemž předpokládám, že tento model nebude obsahovat defekty. Obor detekovaných chyb systému zahrnuje i hardwarové defekty, protože dostupné analýzy nasvědčují tomu, že až třetina hardwarových defektů se projeví chybami vykonávání v software.

Rozborem technické i vědecké literatury v oblasti RTOS z hlediska funkční bezpečnosti, tedy tzv. *safety-critical* systémů, jsem zjistil, že zařízení obsahující RTOS mohou dosahovat až úrovně bezpečnosti SIL3. Stanovením této úrovně a její specifikací se zabývá i pro průmyslová zařízení norma IEC 61508. Pro softwarovou část zařízení neexistuje v této normě žádná metrika, ale norma pouze doporučuje metody a techniky (např. nutnost použití *watchdog* nebo *state-of-the-art* technik) pro příslušné úrovně integrity bezpečnosti. Metriku spolehlivosti norma dovoluje uplatnit na systém jako celek, avšak vyčíslení musí být dle normy provedeno až po definovaných intervalech provozu zařízení (např. 10 let). Norma tedy neobsahuje přímý nástroj na vyčíslení zlepšení bezpečnosti softwarových komponent. Avšak do určení úrovně integrity bezpečnosti se započítává i kategorie diagnostické pokrytí, ve které by se moje práce měla projevit.

Z provedené rešerše inovátorských řešení v oblasti verifikace embedded software za jeho běhu jsem našel několik přístupů podobných mému. Většina těchto přístupů se ale nachází ve fázi konceptu, příp. provedené simulace počítající s omezeními (např. funkčnost pouze pro periodická vlákna). V žádném z těchto přístupů jsem ale nenašel rozpracování příslušné teorie nebo definici modelu aplikace. Také jsem našel již hotová řešení, u kterých ale nebylo možné dohledat popis vnitřní funkce nebo další novější zprávy. Vyvozuji z toho, že tato technika je méně často předmětem zkoumání, resp. rozsáhlého výzkumu.

Rešerší v oblasti formálních metod jsem prostudoval různé metody modelování real-time operačních systémů a jejich aplikací. Z uvedených metod se jeví jako nejvhodnější

implementovat model přímo ve formálním jazyce VHDL, protože záměrem této práce je implementace do hardware. Jelikož ale není ještě teorie modelování pro účely online kontroly řádně definována a popsána, rozhodl jsem se použít Petriho sítě, které jsou již hojně využívány pro modelování a analýzu systémů reálného času.

Z analýzy možností implementace algoritmů do hardware vychází použití FPGA jako nejlepší možnost pro vývoj a ladění algoritmů. Pro simulaci, či dokonce úpravu CPU na hardwarové úrovni je možné s výhodou použít tzv. *softcore*. Pro účely mé práce jsem zvolil Nios II od firmy Altera, který sice nepodporuje neinvazivní monitorování běhu aplikace (jako je např. ARM CoreSight koprocesor), ale mohu vytvořit periferní jednotku připojitelnou na sběrnici (Avalon) tohoto procesoru, a využít tak semi-invazivní metodu monitorování v podobě zasílání zpráv z monitorovaného systému.

Ačkoliv se na první pohled může zdát, že teorie návrhových vzorů moc s funkční bezpečností a RTOS nesouvisí, z provedené rešerše vyplývá, že návrhové vzory jsou jeden ze způsobů, kterým se vědci snaží funkční bezpečnost zlepšit. Naráží ovšem na problém, že modelování a popisování chodu programu pomocí formálních jazyků, resp. znovu-využívání již odzkoušených návrhových vzorů pro řešení technických problémů, není mezi programátory rozšířeno. Optikou návrhových vzorů lze také samotné mnou navrhované řešení pokládat za návrhový vzor, resp. vzor architektury.

Jelikož jsou některé chyby, které by měl navrhovaný algoritmus detekovat, definovány slovně až vágně, vytvořil jsem jednoduchý reálný systém postavený na vývojovém kitu STM32F0 obsahující RTOS a dvě vlákna. Pro tento systém jsem vytvořil čtyři scénáře, do kterých jsem na úrovni aplikace zanesl defekty simulující jevy *deadlock*, *livelock* a *race condition*. Také jsem implementoval semi-invazivní monitorování systému a tedy zaznamenal reálný průběh těchto scénářů. Některá z těchto měření splnila očekávání dle definic jednotlivých chyb, ale u jiných se bohužel projevila nedostatečnost monitorovací metody.

Vlastním řešením výše vytyčených cílů je online kontrolní systém. V této práci jsem řešil jak integraci navženého modulu do cílového systému (tedy možnosti rozhraní), tak hlavně definici kontrolního modulu a jeho implementaci do prostředí hradlových polí. Nejdůležitější část kontrolního modulu tvoří model cílového programu, oproti kterému kontrolní modul vyhodnocuje odchylky. Modelů programu jsem v této práci navrhl více a vybral jsem si model událostí programu (resp. RTOS) na základě registru *Program Counter*.

Tento model jsem definoval jako *kontrolní Petriho síť*, přičemž jsem jako základ použil časovou (barevnou) Petriho síť. Po definici notace jsem také definoval operační sémantiku, aby síť reagovala na vstupy (události a registr *Program counter*) z monitorovaného systému. Poté jsem již implementoval tuto síť, resp. uzel, který se skládá z přechodu a místa, do jazyka VHDL pro integraci do hradlových polí.

Za účelem simulace kontrolního modulu jsem vytvořil *testbench* v prostředí Modelsim obsahující kontrolní modul. Jako stimul jsem použil úsek záznamu z reálného běhu RTOS se dvěma vlákny, který jsem převedl na časovou posloupnost událostí definovaných prostřednictvím hodnoty registru *Program counter*. Simulací jsem tedy ověřil funkčnost konrolního modulu a jeho správnou reakci na jednotlivé třídy situací, čímž je běh neobsahující chybu, běh s časovou chybou a běh s chybou kontinuity událostí. Kontrolní modul detekoval chyby v čase jednoho hodinového cyklu, tj. 10 *ns* od jejich vzniku. Tímto jsem alespoň v laboratorních podmínkách ověřil koncept mého řešení, čímž se řešení dostává na úroveň TRL3.

Abych mohl simulaci provést, musel jsem ještě manuálně specifikovat model programu obsahující platné časové intervaly vykonávání jednotlivých událostí (funkcí) a jejich rozsahy fyzických adres. Model je také připraven na strojové vytvoření, což zapadá do konceptu MDA. Dle tohoto konceptu se nejdříve vytvoří model programu, který bude transformovatelný do *kontrolní Petriho sítě*. K tomu bude nejvhodnější využít návrhové vzory vytvořené z již odzkoušených programových konstrukcí. Následným převedením této sítě do prostředí FPGA by byl kontrolní modul implementovaný.

Literární průzkum i tato práce také poukázala na fakt, že i když je oblast funkční bezpečnosti pro embedded software dlouhodobě zkoumána, nevyvinula se ještě technika, která by pomohla eliminovat nebo detekovat všechny potenciální chyby dané aplikace. Závěrem lze také říci, že tato práce přinesla více otázek a možných směrů dalšího výzkumu, než bylo na počátku.

10.1 Vyhodnocení hypotéz

Hypotéza č. 1: Abych mohl kvantitativně vyjádřit zlepšení diagnostického pokrytí systému za použití navrhnutého kontrolního subsystému, musel bych provést komplexní testování. Vzhledem k tomu, že provedená simulace ukázala, že je subsystém schopen detekovat i drobné časové anomálie (zpoždění vykonávání systémové funkce) v rámci mikrosekund, můžu prohlásit, že navrhnutý subsystém zlepší diagnostické pokrytí.

Hypotéza č. 2: V této práci jsem neprokazoval zlepšení funkční spolehlivosti za použití návrhových vzorů v procesu vývoje systému, neboť takovýto výzkum, který kvantifikuje změnu spolehlivosti bez a s použitím daného návrhového vzoru, již existuje. Tuto skutečnost je obtížné kvantifikovat, proto ostatní literatura uvádí potvrzení pouze kvalitativně.

Hypotéza č. 3: Ano, i když jsou nástroje určené pro modelování konkurentních programů (pro mikroprocesory), lze je po provedení úprav jejich specifikace použít.

K tomuto procesu jsou vhodné pouze některé formální nástroje.

Hypotéza č. 4: Ano, implementace algoritmů v hardware podporuje prostředí hradlových polí, a to i při vývoji prototypů. Implementaci v hardware také doporučuje standard pro funkční bezpečnost tím, že požaduje diverzitu, jednoznačnost a verifikovatelnost bezpečnostních funkcí.

10.2 Přínosy

Kromě přínosů plynoucích z jednotlivých rešerší, které ústí ve vytvoření přehledných pohledů na dané oblasti, sem řadím hlavně následující odvedenou práci:

- Integroval a vyzkoušel jsem dostupné nástroje pro monitorování běhu embedded software, resp. RTOS pro ARM Cortex-M architektury, resp. STM32F0. Jako vhodný nástroj hodnotím knihovnu Segger ve spojení se software SystemView (volný pro nekomerční účely). Za použití debugovacího zařízení J-Link (komerční, lehce dostupný) komunikujícího s Cortex-M pomocí JTAG na frekvenci 1 MHz je komunikace plynulá a nebrzdí vlastní aplikaci. V případě téměř 100 % vytížení aplikace však tento způsob nedostačuje a musí se použít neinvazivní monitorování, např. pomocí zařízení J-Trace (drahý komerční nástroj).
- Specifikoval jsem scénáře aplikací s RTOS obsahující nežádoucí defekty, které se projevují jako chyby výkonávání RTOS (*deadlock*, *livelock* a *starvation*). Tyto scénáře slouží pro ilustraci projevů těchto chyb a také mohou sloužit jako chybné implementace pro vyhodnocení diagnostického pokrytí různých metod a technik. Také z toho vyplývají tyto závěry: a) defekty v software se mohou projevit jako chyby běhu RTOS; b) detekcí anomálií událostí RTOS lze detekovat, a dokonce i klasifikovat chyby RTOS.
- Definoval jsem architekturu subsystému pro kontrolu běhu programu dle jeho modelu. Tento koncept se objevuje ve více technických řešeních, a proto jsem vytvořil definici tohoto vzoru architektury dle dostupných konvencí.
- Popsal jsem vytvořenou kontrolní Petriho síť, u které jsem definoval notační a operační sémantiku. Uvedená definice je rozšířením časové (barevné) Petriho sítě a monitorující Petriho sítě, která již také upravuje operační sémantiku standardní Petriho sítě. Dle této definice jsem implementoval základní prvek tohoto modelu (uzel) do prostředí hradlových polí pomocí jazyka VHDL.
- Vytvořil jsem simulační systém (*testbench*) v jazyce SystemVerilog pro prostředí ModelSim, který obsahuje vytvořený kontrolní modul s modelem demonstrační aplikace a stimul (časová data) zachycující chod demonstrační aplikace na reálné platformě.

10.3 Aplikace

Kromě publikací uvedených níže jsem také již teoretické a praktické poznatky plynoucí z této práce aplikoval při řešení evropského projektu SECREDAS. Konkrétněji se jednalo o vytvoření pokročilejší techniky detekující chyby software v oblasti automotive vestavných systémů. Realizace proběhla v rámci WP3 Task 3.1 a byla zařazena mezi *Common Technology Elements* v Deliverable D3.1.

Aktuálně také probíhá aplikace tohoto kontrolního modulu v oblasti průmyslu, resp. Industry 4.0. Se zvyšujícími se nároky na softwarové vybavení kyber-fyzikálních systémů založených na embedded platformě vzrůstá také požadavek na zajištění adekvátní funkční bezpečnosti. Už se nejedná pouze o detekci chyby, ale i o její klasifikaci a adekvátní reakci systému.

10.4 Směry pro další výzkum

Z provedených rešerší a procesu celé práce plynou další cesty potenciálního výzkumu v různých oblastech. První z těchto oblastí se týká analýzy potenciálních chyb v RTOS. Z rešerše vyplynulo, že výzkum v této oblasti je dlouhodobě aktivní, avšak komplexní analýza a kvantifikace chyb vznikajících v software není dostatečná. Výzkumy se většinou omezují pouze na kategorizaci, což je logická cesta pro snížení komplexity této oblasti. Zajímavým spojením se jeví analýza defektů, resp. chyb a jejich projevů, které je možné měřit, např. pomocí profilace běhu programu. Tím pádem by bylo možné tyto chyby automaticky klasifikovat a strojově na ně reagovat, příp. adaptovat.

Při procesu specifikace a definice modelu programu jsem navrhl koncepty dalších modelů, které popisují program z určitého pohledu, a tedy mají potenciál detekovat jiné typy chyb. Jedná se např. o model profilace běhu programu nebo model běhu všech vláken RTOS aplikace. Rozvinutí teorie pro definici těchto modelů a příslušných technik (např. techniky využívající strojové učení) pro klasifikaci chyb za běhu aplikace skýtá potenciál ke vzniku dalších technik detekce chyb.

VLASTNÍ PUBLIKAČNÍ ČINNOST - VYBRANÉ PUBLIKACE

- ARM, J.; BRADÁČ, Z.; BAŠTÁN, O.; STREIT, J.; MIŠÍK, Š. Design pattern for the runtime model-based checking of a real-time embedded system. In *16th IFAC International Conference on Programmable Devices and Embedded Systems - PDeS 2019.*
- ARM, J.; BRADÁČ, Z.; FIEDLER, P.; KACZMARCZYK, V. Characterizing the Simulink-based Code Generation Toolchain for Safety-critical Applications in an ARM Cortex-R Target. In *16th IFAC International Conference on Programmable Devices and Embedded Systems - PDeS 2019.*
- MARCOŇ, P.; BRADÁČ, Z.; ZEZULKA, F.; ARM, J.; BENEŠL, T. Digital Twin and AAS in the Industry 4.0 Framework. *IOP Conference Series: Materials Science and Engineering*, 2019, roč. 618, č. 1, s. 1–8. ISSN: 1757-899X.
- MARCOŇ, P.; ARM, J.; BENEŠL, T.; ZEZULKA, F.; DOHNAL, P.; DIEDRICH, C.; SCHRÖDER, T.; BELYAEV, A.; KŘÍŽ, T.; BRADÁČ, Z. New Approaches to Implementing the SmartJacket into Industry 4.0. *SENSORS*, 2019, roč. 19, č. 7, s. 1–21. ISSN: 1424-8220.
- ARM, J.; ZEZULKA, F.; BRADÁČ, Z.; MARCOŇ, P.; KACZMARCZYK, V.; BENEŠL, T. Implementing Industry 4.0 in Discrete Manufacturing: Options and Drawbacks. In *15th IFAC Conference on Programmable Devices and Embedded Systems - PDeS 2018. IFAC-PapersOnLine (ELSEVIER).* 2018. s. 1–6. ISSN: 2405-8963.
- ZEZULKA, F.; MARCOŇ, P.; BRADÁČ, Z.; ARM, J.; BENEŠL, T.; VESELÝ, I. Communication Systems for Industry 4.0 and the IIoT. In *15th IFAC Conference on Programmable Devices and Embedded Systems - PDeS 2018. IFAC-PapersOnLine (ELSEVIER).* 2018. s. 1–6. ISSN: 2405-8963.
- MARCOŇ, P.; ZEZULKA, F.; ARM, J.; BENEŠL, T.; VESELÝ, I.; BRADÁČ, Z.; DIEDRICH, C.; SCHRÖDER, T.; BELYAEV, A. The Asset Administration Shell of Operator in the Platform of Industry 4.0. In *Proceedings of the 2018 18th International Conference on Mechatronics – Mechatronika (ME) 2018.* Czech Republic, Brno: IEEE, 2018. s. 559–563. ISBN: 978-80-214-5543-6.
- KACZMARCZYK, V.; BAŠTÁN, O.; BRADÁČ, Z.; ARM, J. An Industry 4.0 Testbed (Self-Acting Barman): Principles and Design. *IFAC-PapersOnLine (ELSEVIER)*, 2018, roč. 51, č. 6, s. 163–270. ISSN: 2405-8963.
- ARM, J.; MIŠÍK, Š.; BRADÁČ, Z.; STREIT, J. CNC Motion Controller Testing Methods. In *15th IFAC Conference on Programmable Devices and Embedded Systems - PDeS 2018. IFAC-PapersOnLine (ELSEVIER).* 2018. s. 1–6. ISSN:

2405-8963.

- ARM, J.; BRADÁČ, Z.; KACZMARCZYK, V. Real-time capabilities of Linux RTAI. In *Proceedings on 14h IFAC Conference on Programmable Devices and Embedded Systems PDES 2016 (Preprint). IFAC-PapersOnLine (ELSEVIER)*. Brno: 2016. s. 446–451. ISBN: 9781510835023. ISSN: 2405-8963.
- ARM, J.; BRADÁČ, Z.; FIEDLER, P. Fault Tolerant CNC Motion Controller. In *Proceedings on 14h IFAC Conference on Programmable Devices and Embedded Systems PDES 2016 (Preprint). IFAC-PapersOnLine (ELSEVIER)*. Brno: 2016. s. 210–215. ISBN: 9781510835023. ISSN: 2405-8963.
- ARM, J.; BRADÁČ, Z. Real-time Virtual Machine Simulator with Collision Detection. In *Sborník příspěvků studentské konference Blansko 2016*. 2016. s. 4–8. ISBN: 978-80-214-5389- 0.
- ARM, J. Web Control of the Robotic Arm. In *Proceedings of the 22nd Conference STUDENT EEICT 2016*. 1. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2016. s. 456–460. ISBN: 978-80-214-5350- 0.
- ARM, J.; BRADÁČ, Z.; ŠTOHL, R. Increasing Safety and Reliability of Roll-back and Roll-forward Lockstep Technique for Use in Real- time Systems. In *Proceedings on 14h IFAC Conference on Programmable Devices and Embedded Systems PDES 2016 (Preprint). IFAC-PapersOnLine (ELSEVIER)*. Brno: 2016. s. 506–511. ISBN: 9781510835023. ISSN: 2405-8963.
- ARM, J. Real-time Crane Control via PC. In *Proceedings of the 21st Conference STUDENT EEICT 2015*. Brno, Česká republika: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2015. s. 440–444. ISBN: 978-80-214-5148- 3.
- ARM, J.; BRADÁČ, Z. Hardware scheduler monitor. In *Sborník příspěvků studentské konference Kohútka 2015*. 2015. s. 4–7. ISBN: 978-80-214-5239- 8.

LITERATURA

[1] Aalst, W. M. P.: Petri net based scheduling. *Operations-Research-Spektrum*, 1996: s. 219–229, doi:10.1007/BF01540160.

[2] Abate, F.; Sterpone, L.; Lisboa, C. A.; aj.: New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors. *IEEE Transactions on Nuclear Science*, 2009: s. 1992–2000, doi:10.1109/TNS.2009.2013237.

[3] Abdul-Hussin, M.: Design of a Petri Net Based Deadlock Prevention Policy Supervisor for S3PR. In *2015 6th International Conference on Intelligent Systems, Modelling and Simulation*, Feb 2015, ISSN 2166-0662, s. 46–52, doi:10.1109/ISMS.2015.54.

[4] Alur, R.; Dill, D. L.: A theory of timed automata. *Theoretical Computer Science*, ročník 126, č. 2, 1994: s. 183–235, ISSN 0304-3975, doi:http://dx.doi.org/10.1016/0304-3975(94)90010-8.
URL http://www.sciencedirect.com/science/article/pii/0304397594900108

[5] Armoush, A.: *Design Patterns for Safety-Critical Embedded Systems*. Dizertační práce, RWTH Aachen University, 2010.

[6] Arora, D.; Raghunathan, A.; Jha, N. K.: Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2007: s. 1295–1308, doi:10.1109/TVLSI.2006.887799.

[7] Atig, M.; Bensalem, S.; Bliudze, S.; aj. (editoři): *Verification and Evaluation of Computer and Communication Systems*, 12th International Conference, VECoS 2018, Grenoble, France, September 26–28, 2018, Proceedings, Springer Nature Switzerland AG, 2018, ISBN 978-3-030-00358-6, doi:https://doi.org/10.1007/978-3-030-00359-3.

[8] Babar, M. I.; Ramzan, M.; Ghayyur, S.: Challenges and future trends in software requirements prioritization. In *Proceedings - International Conference on Computer Networks and Information Technology*, 07 2011, ISBN 978-1-61284-940-9, s. 319–324, doi:10.1109/ICCNIT.2011.6020888.
URL https://www.researchgate.net/publication/261110089_Challenges_and_future_trends_in_software_requirements_prioritization

[9] Baier, C.; Katoen, J.-P.: *Principles of Model Checking*, ročník 26202649. The MIT Press, 01 2008, ISBN 978-0262026499.

[10] Baroni, A. L.; gaël Guéhéneuc, Y.; Albin-amiot, H.: Design Patterns Formalization. 2003.

[11] Beck, K.; Cunningham, W.: Using Pattern Languages for Object Oriented Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987.

[12] Beitollahi, H.; Deconinck, G.: Fault-Tolerant Rate-Monotonic Scheduling Algorithm in Uniprocessor Embedded Systems. *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006: s. 395–396, doi:10.1109/PRDC.2006.35.

[13] Bernard, B.: *Vývojové prostředí pro návrhové vzory v C#*. Diplomová práce, Vysoká škola ekonomická v Praze, 2006.

[14] Bernardi, S.; Campos, J.: Computation of Performance Bounds for Real-Time Systems Using Time Petri Nets. *IEEE Transactions on Industrial Informatics*, 2009: s. 168–180, doi:10.1109/TII.2009.2017201.

[15] Blouin, D.; Chillet, D.; Senn, E.; aj.: AADL Extension to Model Classical FPGA and FPGA Embedded within a SoC. *International Journal of Reconfigurable Computing*, 2011, doi:10.1155/2011/425401.
URL https://hal.inria.fr/hal-00650628

[16] Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, ročník 25, č. 6, Nov 2005: s. 10–16, ISSN 1937-4143, doi:10.1109/MM.2005.110.

[17] Bosch, J.: Design Patterns as Language Constructs. *Journal of Object Oriented Programming*, ročník 11, 12 1996.

[18] Boukhelfa, K.; Belala, F.: Towards a Formalization of Real-Time Patterns-Based Designs. In *Computer Science and Its Applications*, editace A. Amine; L. Bellatreche; Z. Elberrichi; E. J. Neuhold; R. Wrembel, Cham: Springer International Publishing, 2015, ISBN 978-3-319-19578-0, s. 624–635.

[19] Bruintjes, H.; Katoen, J. P.; Lesens, D.: A Statistical Approach for Timed Reachability in AADL Models. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, ISSN 1530-0889, s. 81–88, doi:10.1109/DSN.2015.32.

[20] Buhr, R. J. A.: *Systems Design with ADA*. USA: Prentice-Hall, Inc., 1984, ISBN 0138816239.

[21] Buschmann, F.; Meunier, R.; Rohnert, H.; aj.: *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996, ISBN 978-0471958697.

[22] Butazzo, G. C.: *Predictable Scheduling Algorithms and Applications*. Springer, 2011, ISBN 978-1-4614-0675-4.

[23] Buttazzo, G.: Real-Time Operating Systems: Problems and Novel Solutions. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, editace W. Damm; E. R. Olderog, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, ISBN 978-3-540-45739-8, s. 37–51.

[24] Béchennec, J.; Faucou, S.; Roux, O. H.; aj.: Testing Real-Time Systems With Runtime Enforcement. *IEEE Design Test*, ročník 35, č. 4, Aug 2018: s. 31–37, ISSN 2168-2356, doi:10.1109/MDAT.2018.2791801.

[25] Carnevali, L.; Melani, A.; Santinelli, L.; aj.: Probabilistic Deadline Miss Analysis of Real-Time Systems Using Regenerative Transient Analysis. Technická zpráva, University of Florence, 2014.

[26] Christopher Alexander et. al.: *A Pattern Language*. Oxford University Press, 1977.

[27] Chupilko, M. M.; Kamkin, A. S.: Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces. In *Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013.*, 2013, s. 67–81, doi:10.4204/EPTCS.111.6.
URL https://doi.org/10.4204/EPTCS.111.6

[28] Cobham Gaisler AB: Firemní literatura. Technická zpráva, Cobham Gaisler AB, 2018.
URL http://www.gaisler.com/

[29] Colombo, C.; Leucker, M.: 18th International Conference, RV 2018. In *Runtime Verification*, 2018.

[30] Commission, I. E.: 61508 functional safety of electrical/electronic/programmable electronic safety-related systems. 1998, doi:10.3403/03263848.
URL https://webstore.iec.ch/publication/22273

[31] Cooper, S.: Introduction To The VHDL-AMD Modeling Language. Technická zpráva, Mentor Graphics, 2007.

[32] Cortes, L. A.: *A Petri Net based Modeling and Verification Technique for Real-Time Embedded Systems*. Dizertační práce, School of Engineering at Linköping University, 2001.

[33] Dalinger, I.: *Formal verification of a processor with memory management units*. Dizertační práce, Universitas Saraviensis, 2006.

[34] Davidrajuh, R.: Design Issues in Developing a Real-Time Control Simulator. In *2011 UKSim 5th European Symposium on Computer Modeling and Simulation*, Nov 2011, s. 156–161, doi:10.1109/EMS.2011.90.

[35] Delange, J.; Feiler, P.: Architecture Fault Modeling with the AADL Error-Model Annex. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug 2014, ISSN 1089-6503, s. 361–368, doi:10.1109/SEAA.2014.20.

[36] Delange, J.; Hugues, J.: Safety Analysis with AADL. Technická zpráva, Carnegie Mellon University, 2015.

[37] Dong, J.: UML Extensions for Design Pattern Compositions. *Journal of Object Technology*, ročník 1, 11 2002: s. 151–163, doi:10.5381/jot.2002.1.5.a3.

[38] Dong, Y.; Wang, G.; Zhao, H.: A Model-Based Testing for AADL Model of Embedded Software. *2009 9th International Conference on Quality Software (QSIC 2009)*, ročník 00, č. undefined, 2009: s. 185–190, ISSN 1550-6002, doi: doi.ieeecomputersociety.org/10.1109/QSIC.2009.33.

[39] Douglass, B. P.: Doing real-time UML systems design using the Harmony Process. In *Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems Development*, 2007.
URL https://www.embedded.com/design/prototyping-and-development/4007239/3/Doing-real-time-UML-systems-design-using-the-Harmony-process-Part-1

[40] Douglass, B. P.: *Design Patterns for Embedded Systems in C*. Newnes, 2011, doi:https://doi.org/10.1016/C2009-0-19352-2.

[41] Douglass, B. P.; Page, P. D.: Real-Time Design Patterns. 1998.

[42] DPAToolkit: DPAToolkit. 2008.

[43] Dubrova, E.: *Fault-Tolerant Design*. Springer-Verlag New York, 2013, doi:10.1007/978-1-4614-2113-9.

[44] Eden, A.; yossi Gil, J.; Yehudai, A.: Automating the Application of Design Patterns. *Journal of Object-oriented Programming*, ročník 10, 11 1997.

[45] Eden, A. H.; Hirshfeld, Y.; Lundqvist, K.: LePUS–symbolic logic modeling of object oriented architectures: A case study. In *Second Nordic Workshop on Software Architecture-NOSA'99*, 1999.

[46] Center for Electronic Systems Design, B.: Fundamental Algorithms for System Modeling, Analysis, and Optimization. 2016.
URL `https://ptolemy.berkeley.edu/projects/embedded/eecsx44/`

[47] Fang, L.; Kitamura, T.; Do, T. B. N.; aj.: Formal Model-Based Test for AUTOSAR Multicore RTOS. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012: s. 251–259, doi:`10.1109/ICST.2012.105`.

[48] Feiler, P. H.; Greenhouse, A.: OSATE Plug-in Development Guide. Technická zpráva, Carnegie Mellon University, 2005.

[49] Frame, A.; Turner, C.; ARM: Introducing New ARM Cortex-R Technology for Safe and Reliable Systems. *Embedded World Conference 2011. Nuremberg*, 2011.

[50] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0-201-63361-2.

[51] Garg, A.: Soft Error Fault Tolerant Systems: CS456 Survey. [online], 2005.
URL `http://www.ece.rochester.edu/~garg/documents/garg.cs456survey05.pdf`

[52] Ghezzi, C.; Mandrioli, D.; Morasca, S.; aj.: A unified high-level Petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, ročník 17, č. 2, Feb 1991: s. 160–172, ISSN 0098-5589, doi:`10.1109/32.67597`.

[53] Giese, H.; Karsai, G.; Lee, E. A.; aj.: *Model-Based Engineering of Embedded Real-Time Systems*, *Lecture Notes in Computer Science*, ročník 6100. Springer, 2010.
URL `http://chess.eecs.berkeley.edu/pubs/847.html`

[54] Gomes, L.; Costa, A.; Barros, J. P.; aj.: From Petri net models to VHDL implementation of digital controllers. In *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*, Nov 2007, ISSN 1553-572X, s. 94–99, doi:`10.1109/IECON.2007.4460403`.

[55] Gomez-Cornejo, J.; Zuloaga, A.; Kretzschmar, U.; aj.: Fast context reloading lockstep approach for SEUs mitigation in a FPGA soft core processor. *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*, 2013: s. 2261–2266, doi:10.1109/IECON.2013.6699483.

[56] Govindarajan, R.; Suciu, F.; Zuberek, W. M.: Timed Petri net models of multithreaded multiprocessor architectures. In *Proceedings of the Seventh International Workshop on Petri Nets and Performance Models*, June 1997, ISSN 1063-6714, s. 153–162, doi:10.1109/PNPM.1997.595546.

[57] Greb, K.; Pradhan, D.: Hercules Microcontrollers: Real-time MCUs for safety-critical products. *Texas Instruments*, 2011.
URL http://www.ti.com/

[58] Greenfield, J.; Short, K.: Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. In *Generative Programming and Component Engineering*, ročník 3154, 10 2003, doi:10.1145/949344.949348.

[59] Group, O. M.: Model Driven Architecture. 2018.
URL https://www.omg.org/mda/

[60] Group, O. M.: The Unified Modeling Language. 2018.
URL https://www.uml-diagrams.org/

[61] Guihal, D.; Andrieux, L.; Esteve, D.; aj.: VHDL-AMS Model Creation. In *Proceedings of the International Conference Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006.*, June 2006, ISSN null, s. 549–554, doi: 10.1109/MIXDES.2006.1706640.

[62] Hedin, G.: Language Support for Design Patterns Using Attribute Extension. In *Object-Oriented Technologys*, editace J. Bosch; S. Mitchell, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, ISBN 978-3-540-69687-2, s. 137–140.

[63] Hesselink, W.; IJbema, M.: Starvation-free mutual exclusion with semaphores. *Formal Aspects of Computing*, ročník 25, 11 2013, doi:10.1007/s00165-011-0219-y.

[64] Hugues, J.; Zalila, B.; Pautet, L.; aj.: From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Trans. Embed. Comput. Syst.*, ročník 7, č. 4, 2008: s. 42:1–42:25, ISSN 1539-9087, doi: 10.1145/1376804.1376810.
URL http://doi.acm.org/10.1145/1376804.1376810

[65] Husák, M.: Spolehlivost systémů. Technická zpráva, České Vysoké Učení Technické, 2014.
URL `http://www.micro.feld.cvut.cz/home/x34ezs/prednasky/12%20Spolehlivost%20systemu.pdf`

[66] Ignat, N.; Nicolescu, B.; Savaria, Y.; aj.: Soft-error classification and impact analysis on real-time operating systems. *Proceedings of the Design Automation & Test in Europe Conference*, 2006, doi:`10.1109/DATE.2006.244063`.

[67] International, S.: Architecture Analysis and Design Language. 2012.
URL `http://www.aadl.info/aadl/currentsite/`

[68] Izosimov, V.; Pop, P.; Eles, P.; aj.: Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems. *Design, Automation and Test in Europe*, ročník 2, 2005: s. 864–869, doi:`10.1109/DATE.2005.116`.

[69] Jakšič, S.; Bartocci, E.; Grosu, R.; aj.: From signal temporal logic to FPGA monitors. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Sep. 2015, ISSN null, s. 218–227, doi:`10.1109/MEMCOD.2015.7340489`.
URL `https://doi.org/10.1109/MEMCOD.2015.7340489`

[70] Jones, M. B.: What really happened on Mars?
URL `http://research.microsoft.com/~mbj/Mars_Pathfinder/`

[71] Kahl, W.: Chapter 9 Labelled Transition Systems. online, 2007.
URL `https://www.cas.mcmaster.ca/~kahl/SE3BB4/2007/SE3B-2007-LTS1_4up.pdf`

[72] Kahl, W.: Software Design III - Concurrent System Design. online, 2007.
URL `https://www.cas.mcmaster.ca/~kahl`

[73] KEIL, A.: CoreSight Technology. online, 2018.
URL `http://www.keil.com/support/man/docs/ulinkpro/ulinkpro_cs_core_sight.htm`

[74] Koopman, P.: A Case Study of Toyota Unintended Acceleration and Software Safety. November 2014.
URL `http://chess.eecs.berkeley.edu/pubs/1081.html`

[75] Krizan, J.; Ertl, L.; Bradac, M.; aj.: Automatic code generation from Matlab/Simulink for critical applications. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, May 2014, ISSN 0840-7789, s. 1–6, doi:`10.1109/CCECE.2014.6901058`.

[76] Lahiri, S.; Wang, C. (editoři): *Automated Technology for Verification and Analysis*, 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings, Springer Nature Switzerland AG, 2018, ISBN 978-3-030-01089-8, doi:https://doi.org/10.1007/978-3-030-01090-4.

[77] Laplante, P. A.: *Real-time Systems Design and Analysis*. John Wiley & Sons, Inc., 2004, ISBN 0-471-22855-9.

[78] Last, M.; Eyal, S.; Kandel, A.: *Effective Black-Box Testing with Genetic Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-32605-2, s. 134–148, doi:10.1007/11678779_10.
URL 10.1007/11678779_10

[79] Lee, J.: *Hardware/Software Deadlock Avoidance for Multiprocessor Multiresource System-on-a-Chip*. Dizertační práce, GeorgiaTech, 12 2004.

[80] Lewis, B.: The Emerging SAE AADL Standard: An Architecture Analysis & Design Language for Building Embedded Real-Time Systems. Technická zpráva, Society of Automotive Engineers, 2018.

[81] Li, Q.; Yao, C.: *Real-time concepts for embedded systems*. CMP Books, 2003, ISBN 978-1-57820-124-2.

[82] Lime, D.; Roux, O.; Jard, C.: Clock Transition Systems. *CEUR Workshop Proceedings*, ročník 928, 08 2012.

[83] Lime, D.; Roux, O. H.: Formal verification of real-time systems with preemptive scheduling. *Real-Time Systems*, ročník 41, 2009: s. 118–151, doi:10.1007/s11241-008-9059-0.

[84] Lips, R.: RTOS for Safety-Related Systems. *IAR Systems*, 2018.
URL https://www.iar.com/support/resources/articles/rtos-for-safety-related-systems/

[85] Lyons, W.: Enabling Increased Safety with Fault Robustness in Microcontroller Applications. *ARM*, 2008.
URL http://www.smarterworld.de/fileadmin/media/whitepaper/files/Enabling_Increased_Safety_with_Fault_Robustness_in_MCU_Applications.pdf

[86] Mak, J. K. H.; Choy, C. S. T.; Lun, D. P. K.: Precise modeling of design patterns in UML. In *Proceedings. 26th International Conference on Software Engineering*, May 2004, ISSN 0270-5257, s. 252–261, doi:10.1109/ICSE.2004.1317447.

[87] Maler, O.; Nickovic, D.; Pnueli, A.: Real Time Temporal Logic: Past, Present, Future. In *Formal Modeling and Analysis of Timed Systems*, editace P. Pettersson; W. Yi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ISBN 978-3-540-31616-9, s. 2–16.

[88] Malm, T.; Vuori, M.; Rauhamäki, J.; aj.: *Safety-critical software in machinery applications*. číslo 2601 in VTT Tiedotteita - Meddelanden - Research Notes, Finland: VTT Technical Research Centre of Finland, 2011.

[89] Matyáš, J.: *Překladač jazyka VHDL pro potřeby formální verifikace*. Diplomová práce, Vysoké učení technické v Brně, 2015.
URL `http://hdl.handle.net/11012/52489`

[90] Mediouni, B. L.; Nouri, A.; Bozga, M.; aj.: BIP 2.0: Statistical Model Checking Stochastic Real-Time Systems. In *Automated Technology for Verification and Analysis*, editace S. K. Lahiri; C. Wang, Cham: Springer International Publishing, 2018, ISBN 978-3-030-01090-4, s. 536–542.

[91] Mejia-Alvarez, P.; Mosse, D.: A responsiveness approach for scheduling fault recovery in real-time systems. *Real-Time Technology and Applications Symposium*, 1999: s. 4–13, doi:`10.1109/RTTAS.1999.777656`.

[92] Mens, T.; Eden, A. H.: On the Evolution Complexity of Design Patterns. *Electronic Notes in Theoretical Computer Science*, ročník 127, č. 3, 2005: s. 147–163, ISSN 1571-0661, doi:`https://doi.org/10.1016/j.entcs.2004.08.041`, proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004).
URL `http://www.sciencedirect.com/science/article/pii/S1571066105001465`

[93] Mistry, J.: *FreeRTOS and Multicore*. Dizertační práce, University of York, 2011.

[94] M.Osmar; Wellings, A.: Run Time Detection of Blocking Time Violations in Real-Time Systems. *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008, doi:`10.1109/RTCSA.2008.39`.

[95] Mosse, D.; Melhem, R.; Ghosh, S.: A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Transactions on Software Engineering*, 2003: s. 752–767, doi:`10.1109/TSE.2003.1223648`.

[96] Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 2002: s. 541–580, doi:`10.1109/5.24143`.

[97] Murata, T.; Shenker, B.; Shatz, S. M.: Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, ročník 15, č. 3, March 1989: s. 314–326, ISSN 2326-3881, doi:10.1109/32.21759.

[98] Murphy, N.: Watchdog timers. *EE Times-India*, 2000.

[99] Nicholson, J.; Eden, A. H.; Gasparis, E.; aj.: Automated verification of design patterns: A case study. *Science of Computer Programming*, ročník 80, 2014: s. 211–222, ISSN 0167-6423, doi:https://doi.org/10.1016/j.scico.2013.05.007.
URL http://www.sciencedirect.com/science/article/pii/S0167642313001354

[100] Oliveira, D.; Matos, R.; Dantas, J.; aj.: Advanced Stochastic Petri Net Modeling with the Mercury Scripting Language. In *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS 2017, New York, NY, USA: Association for Computing Machinery, 2017, ISBN 9781450363464, s. 192–197, doi:10.1145/3150928.3150959.
URL https://doi.org/10.1145/3150928.3150959

[101] Palopoli, L.; Buttazzo, G.; Ancilotti, P.: A C language extension for programming real-time applications. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*, 1999, s. 103–110.

[102] Peres, L.; Künzle, L.; Todt, E.: Applying global time petri net analysis on the Embedded Software context. *Sba: Controle & Automação Sociedade Brasileira de Automatica*, ročník 22, 12 2011: s. 610–619, doi:10.1590/S0103-17592011000600006.

[103] Pham, H.-M.; Pillement, S.; Piestrak, S. J.: Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor. *IEEE Transactions on Computers*, 2012: s. 1179–1192, doi:10.1109/TC.2012.55.

[104] Pohronská, M.; Krajčovič, T.: FPGA Implementation of Multiple Hardware Watchdog Timers for Enhancing Real-Time Systems Security. *EUROCON - International Conference on Computer as a Tool*, 2011: s. 1–4, doi:10.1109/EUROCON.2011.5929215.

[105] Rauhamäki, J.: *Designing Functional Safety Systems: A Pattern Language Approach*. Dizertační práce, TampereUniversity of Technology, 2017.

[106] Reisner, L.: Operační systémy reálného času s Win32 API. In *Automa*, Automa – časopis pro automatizační techniku, 2017.
URL http://automa.cz/cz/casopis-clanky/operacni-systemy-realneho-casu-s-win32-api-2003_03_28762_3569/

[107] Reorda, M. S.; Violante, M.; Meinhardt, C.; aj.: A low-cost SEE mitigation solution for soft-processors embedded in Systems on Pogrammable Chips. *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009: s. 352–357, doi:10.1109/DATE.2009.5090687.

[108] Rhoads, S.: Plasma CPU. online.
URL http://plasmacpu.no-ip.org/

[109] Rokyta, P.; Fengler, W.; Hummel, T.: *Electronic System Design Automation Using High Level Petri Nets*. Boston, MA: Springer US, 2000, ISBN 978-1-4757-3143-9, s. 193–204, doi:10.1007/978-1-4757-3143-9_10.
URL https://doi.org/10.1007/978-1-4757-3143-9_10

[110] Rollins, N. H.; Wirthlin, M. J.: Reliability of a Softcore Processor in a Commercial SRAM-based FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1155-7, s. 171–174, doi:10.1145/2145694.2145723.
URL http://doi.acm.org/10.1145/2145694.2145723

[111] Roscoe, A. J.; Blair, S. M.; Burt, G. M.: Benchmarking and optimisation of Simulink code using Real-Time Workshop and Embedded Coder for inverter and microgrid control applications. In *2009 44th International Universities Power Engineering Conference (UPEC)*, Sep. 2009, ISSN null, s. 1–5.

[112] Roux, O. H.; Déplanche, A.-M.: A T-time Petri net extension for real time-task scheduling modeling. *European Journal of Automation, Hermés Science*, 2010: s. 973–987.

[113] Ryu, M.; Kim, S.-J.: Deterministic and Statistical Deadline Guarantees for a Mixed Set of Periodic and Aperiodic Tasks. Technická zpráva, Seoul National University, 2000.

[114] Safarulla, I. M.; Manilal, K.: Design of Soft error tolerance technique for FPGA based soft core processors. *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*, 2014, doi:10.1109/ICACCCT.2014.7019254.

[115] Schlaepfer, E.: Comparison of Internal and External Watchdog Timers. *Maxim Integrated*, 2008.

[116] Schwaderer, C.: RTOS fault tolerance, error detection, and correction. *Microware Systems Corporation*, 2015.

[117] Shah, S. M.; Irfan, M.: Embedded hardware/software verification and validation using hardware-in-the-loop simulation. In *Proceedings of the IEEE Symposium on Emerging Technologies, 2005.*, Sep. 2005, ISSN null, s. 494–498, doi:10.1109/ICET.2005.1558931.

[118] Shi, J.; Zhu, L.; Huang, Y.; aj.: Binary Code Level Verification for Interrupt Safety Properties of Real-Time Operating System. *Theoretical Aspects of Software Engineering (TASE)*, 2012: s. 223–226, doi:10.1109/TASE.2012.46.

[119] Silva, D.; Bolzani, L.; Vargas, F.: An intellectual property core to detect task schedulling-related faults in RTOS-based embedded systems. *2011 IEEE 17th International On-Line Testing Symposium*, 2011: s. 19–24, doi:10.1109/IOLTS.2011.5993805.

[120] Smith, D. J.; Simpson, K. G.: *Safety Critical Systems Handbook A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849*. Elsevier Ltd., 2011, doi:https://doi.org/10.1016/C2010-0-65791-9.

[121] Soininen, J.-P.; Huttunen, T.; Tiensyrja, K.; aj.: Cosimulation of real-time control systems. *Design Automation Conference, 1995, with EURO-VHDL, Proceedings EURO-DAC '95., European*, 1995: s. 170–175, doi:10.1109/EURDAC.1995.527404.

[122] Soto, E.; Pereira, M.: *Implementing a Petri Net Specification in a FPGA Using VHDL*. Boston, MA: Springer US, 2005, ISBN 978-0-387-28327-2, s. 167–174, doi:10.1007/0-387-28327-7_14.
URL https://doi.org/10.1007/0-387-28327-7_14

[123] Soukup, P.: *Hardwarová realizace Petriho sítí*. Dizertační práce, České vysoké učení technické v Praze, 2011.

[124] Spivey, J. M.: *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989, ISBN 0-13-983768-X.

[125] Springintveld, J.; Vaandrager, F.; D'Argenio, P. R.: Testing timed automata. *Theoretical Computer Science*, ročník 254, č. 1, 2001: s. 225–257, ISSN 0304-3975, doi:http://dx.doi.org/10.1016/S0304-3975(99)00134-6.

URL http://www.sciencedirect.com/science/article/pii/S0304397599001346

[126] Srovnal, V.: Přehled operačních systémů reálného času. *AT&P journal*, 2005.

[127] Stoyenko, A. D.; Halang, W. A.: Extending Pearl for industrial real-time applications. *IEEE Software*, ročník 10, č. 4, July 1993: s. 65–74, ISSN 1937-4194, doi:10.1109/52.219619.

[128] Strejček, J.: *Linear Temporal Logic: Expressiveness and Model Checking*. Dizertační práce, Masaryk University, 2005.

[129] Strnadel, J.; Slimařík, F.: Impact of Software Fault Tolerance to Fault Effects in OS-Driven RT Systems. *Computing and Informatics*, ročník 33, č. 4, 2014: s. 757–782, ISSN 1335-9150.
URL http://www.fit.vutbr.cz/research/view_pub.php.en?id=10340

[130] Sugasawa, Y.; Jin, Q.; Seya, K.: Extended stochastic Petri Net models for systems with parallel and cooperative motions. *Computers & Mathematics with Applications*, ročník 24, č. 1, 1992: s. 119–126, ISSN 0898-1221, doi: https://doi.org/10.1016/0898-1221(92)90236-B.
URL http://www.sciencedirect.com/science/article/pii/089812219290236B

[131] Systems, S.: Enterprise Architect.
URL https://sparxsystems.com/products/ea/index.html

[132] L'Institut de Recherche en Informatique et Systèmes Aléatoires, R.: Initiation à la Vérification. 2009.

[133] Tarrillo, J.; Bolzani, L.; Vargas, F.: A Hardware-Scheduler for Fault Detection in RTOS-Based Embedded Systems. *12th Euromicro Conference on Digital System Design*, 2009, doi:10.1109/DSD.2009.224.

[134] Tempelmeier, T.: On The Real Value Of New Paradigms. 01 2001.

[135] Tichý, J.: Návrhové vzory v programování. SlideShare, 2008.

[136] Todman, T.; Stilkerich, S.; Luk, W.: In-circuit temporal monitors for runtime verification of reconfigurable designs. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, ISSN 0738-100X, s. 1–6, doi:10.1145/2744769.2744856.

[137] Uher, J.: Úvod do funkční bezpečnosti I: norma ČSN EN 61508. *Automa*, ročník 8, 2004.
URL http://automa.cz/index.php?id_document=32520

[138] Vankeirsbilck, J.; Hallez, H.; Boydens, J.: Soft Error Protection in Safety Critical Embedded Applications: An Overview. In *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, Nov 2015, s. 605–610, doi:10.1109/3PGCIC.2015.89.

[139] Wang, J.; Deng, Y.; Xu, G.: Reachability analysis of real-time systems using time Petri nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, ročník 30, č. 5, Oct 2000: s. 725–736, ISSN 1083-4419, doi:10.1109/3477.875448.

[140] Wang, P.: Chapter 10 - Formal Model Based Safety Analysis Methods and the Application. In *Civil Aircraft Electrical Power System Safety Assessment*, editace P. Wang, Butterworth-Heinemann, 2017, ISBN 978-0-08-100721-1, s. 259–287, doi:https://doi.org/10.1016/B978-0-08-100721-1.00010-8.
URL http://www.sciencedirect.com/science/article/pii/B9780081007211000108

[141] Wellhausen, T.; Fiesser, A.: How to Write a Pattern? A Rough Guide for First-Time Pattern Authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*, EuroPLoP '11, New York, NY, USA: Association for Computing Machinery, 2011, ISBN 9781450313025, doi:10.1145/2396716.2396721.
URL https://doi.org/10.1145/2396716.2396721

[142] Xilinx Inc.: Firemní literatura. 2018.
URL https://www.xilinx.com

[143] Xu, D.; He, X.; Deng, Y.: Compositional Schedulability Analysis of Real-Time Systems Using Time Petri Nets. *IEEE Transactions on Software Engineering*, 2002: s. 984–996, doi:10.1109/TSE.2002.1041054.

[144] Zabihi, M.; Farbeh, H.; Miremadi, S. G.: A partial task replication algorithm for fault- tolerant FPGA-based soft-multiprocessors. *Real-Time and Embedded Systems and Technologies (RTEST), 2015 CSI Symposium on*, 2015: s. 1–7, doi:10.1109/RTEST.2015.7369842.

[145] Zezulka, F.; Bradáč, Z.: Development in safety and reliability of programmable devices and systems. *Programmable devices and systems*, 2003.

[146] Zhang, H.; Gu, M.; Sun, J.: Modeling a Heterogeneous Embedded System in Coloured Petri Nets. *Journal of Applied Mathematics*, ročník 2014, 03 2014, doi:10.1155/2014/943094.

[147] Zhao, Y.; Rammig, F.: Model-based Runtime Verification Framework. *Electronic Notes in Theoretical Computer Science*, ročník 253, č. 1, 2009: s. 179–193, ISSN 1571-0661, doi:https://doi.org/10.1016/j.entcs.2009.09.035, proceedings of the Sixth International Workshop on Formal Engineering approches to Software Components and Architectures (FESCA 2009).
URL http://www.sciencedirect.com/science/article/pii/S1571066109003892

[148] Zheng, X.; Julien, C.; Podorozhny, R.; aj.: BraceAssertion: Runtime Verification of Cyber-Physical Systems. In *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*, Oct 2015, s. 298–306, doi:10.1109/MASS.2015.15.

[149] Zhu, Y.; Li, Y.; Xue, J.; aj.: What Is System Hang and How to Handle It. *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, doi:10.1109/ISSRE.2012.12.

[150] Zhu, Z.: Study and Application of Patterns in Software Reuse. In *2009 IITA International Conference on Control, Automation and Systems Engineering (case 2009)*, July 2009, s. 550–553, doi:10.1109/CASE.2009.136.

[151] Čeleda, P.: *Zvýšení spolehlivosti a diagnostika operačních systémů pracujících v reálném čase*. Dizertační práce, Univerzita obrany, 2007.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

CPU centrální procesorová jednotka – Central Processing Unit

FPU výpočetní jednotka – Floating Point Unit

ARM procesor s redukovanou instrukční sadou – Advanced RISC Machines

fRCPU monitorovací periferie pro jádro procesoru firmy Yogitech – faultRobust-CPU

EUC kontrolovaná jednotka – Equipment Under Control

fRDI monitorovací rozhraní – faultRobust Diagnostic Interface

SIL úroveň funkční bezpečnosti – Safety Integrity Levels

SFF zlomek bezpečných chyb – Safe Failure Fraction

MTBF průměrná doba bezporuchového provozu – Mean Time Between Failures

LBIST diagnostika CPU na úrovni transistorů – Logic Built-In Self Test

ECC algoritmus opravování jednobitových chyb v datech – Error Correcting Code

EMI elektromagnetické rušení – Electromagnetic Interference

TMR trojitě redundantní systém – Triple Modular Redundancy

FCCU jednotka vyhodnocení chyby systému – Fault Collection and Control Unit

RTOS operační systém reálného času – Real-time Operating System

VHDL programovací jazyk pro popis hardware – VHSIC Hardware Description Language

MMU jednotka spravující paměťová úložiště – Memory Management Unit

MPU jednotka spravující přístup do paměti – Memory Protection Unit

HAL vrstva software zpřístupňující hardware systému – Hardware Abstraction Layer

RAM paměť s přímým přístupem – Random Access Memory

BORPH hardwarový real-time operační systém vyvinutý na Berkeley – Berkeley Operating system for ReProgrammable Hardware

PLC řídicí prvek průmyslové automatizace – Programmable Logic Controller

SEU přechodná chyba – Single-Event Upsets

OS operační systém – Operating System

RTAI real-time patch pro Linux – Real Time Application Interface

RMS statický periodický plánovací algoritmus – Rate Monotonic Scheduling

EDF dynamický plánovací algoritmus dle deadlinů – Earliest Deadline First

FCFS zařazování dle příchozího pořadí – First Comes First Serves

CRC kontrola integrity dat – Cyclic Redundant Control

EMI reakce na elektromagnetické záření – Electromagnetic Interference

MDA metodika systémového návrhu software dle modelu – Model Driven Architecture

LTL lineární temporální logika – Linear Temporal Logic

CTL logika stromu – Computation Tree Logic

MTL metrická temporální logika – Metric Temporal Logic

GPIO periferní jednotka pro ovládání vstupů a výstupů – General-Purpose Input/Output

CRC kontrolní součet dat – Cyclic Redundant Control

ALU aritmeticko-logická jednotka procesoru – Arithmetic Logic Unit

NVIC jednotka pro zpracování přerušení – Nested Vector Interrupt Control

RESO kontrola výpočtu na základě upravených vstupů – Recomputing with Shifted Operands

POSIX standard ISO/IEC 9945 pro rozhraní funkcí operačního systému – Portable Operating System Interface

PIT programovatelný hardwarový časovač – Programmable Interval Timer

RTC časovač pro světový čas – Real-Time Clock

HPET vysoce přesný časovač – High Precision Event Timer

SIF bezpečnostní funkce – Safety Instrumented Function

NUMA škálovatelná počítačová platforma – Non-uniform Memory Access

JTAG standard pro testování zařízení – Joint Test Action Group

AMBA rodina sběrnic architektury ARM – Advanced Microcontroller Bus Architecture

AHB vysokorychlostní sběrnice procesoru – AMBA High-performance Bus

APB periferní sběrnice procesoru – Advanced Peripheral Bus

AXI sběrnice procesoru pro externí komponenty – Advanced Extensible Interface

PCI sběrnice pro připojení k základní desce – Peripheral Component Interconnect

UART univerzální sběrnice mikrokontroléru – Universal Asynchronous Receiver-Transmitter

SPI sériová sběrnice pro periferie – Serial Peripheral Interface

IP komerční uzavřená komponenta – Intellectual Property

IEC mezinárodní elektrotechnická komise – International Electrotechnical Commission

ISO mezinárodní organizace pro normalizaci – International Organization for Standardization

FPGA programovatelné hradlové pole – Field Programmable Gate Array

ASIC zákaznický integrovaný obvod – Application Specific Integrated Circuit

SoC heterogenní integrovaný obvod – System On the Chip

HPS procesorové jádro – Hardware Processor System

RTL registrová úroveň abstrakce – Register-Transfer Level

HLS syntéza z vyššího jazyka – High-Level Synthesis

MIPS procesor bez automaticky organizované pipeline – Microprocessor without Interlocked Pipeline Stages

DMIPS Dhrystone výkon procesoru – Dhrystone Million Instructions Per Second

UML modelovací jazyk – Unified Modeling Language

TRL úroveň připravenosti technologie – Technology Readiness Level

AADL jazyk popisující architekturu a design – Architecture Analysis & Design Language

FMS strojová výroba – Flexible Manufacturing System

WCET nejhorší doba vykonání – Worst-Case Execution time

CASE nástroje pro automatickou tvorbu software – Computer Aided Software Engineering

FSM konečný stavový automat – Finite State Machine

ETM trasovací subsystém – Embedded Trace Macrocell

ITM uživatelský trasovací subsystém – Instrumentation Trace Macrocell

IDE vývojové prostředí – Integrated Development Environment

CCF porucha se společnou příčinou – Common Cause Failure

SEZNAM PŘÍLOH

A Přílohy **115**

A.1 Implementace uzlu (místo + hrana) kontrolního modelu ve VHDL . 115
A.2 Integrace procesu vytvoření kontrolního systému do standardního V-modelu . . . 116
A.3 Návrh hardware platformy pro měření událostí pomocí mapované periferie . . . 117
A.4 Implementace scénáře 1 – *deadlock* . . . 118
A.5 Záznam událostí RTOS ze scénáře 1 – *deadlock* . . . 119
A.6 Implementace scénáře 2 – *livelock* . . . 120
A.7 Záznam událostí RTOS ze scénáře 2 – *livelock* . . . 121
A.8 Implementace scénáře 3 – *starvation* . . . 122
A.9 Záznam událostí RTOS ze scénáře 3 – starvation . . . 123
A.10 Implementace scénáře 4 – *race condition* . . . 124
A.11 Záznam událostí RTOS ze scénáře 4 – *race condition* . . . 125
A.12 Záznam událostí simulace třídy I – normální běh . . . 126
A.13 Záznam událostí simulace třídy II – časová chyba . . . 127
A.14 Záznam událostí simulace třídy III – chyba kontinuity . . . 128

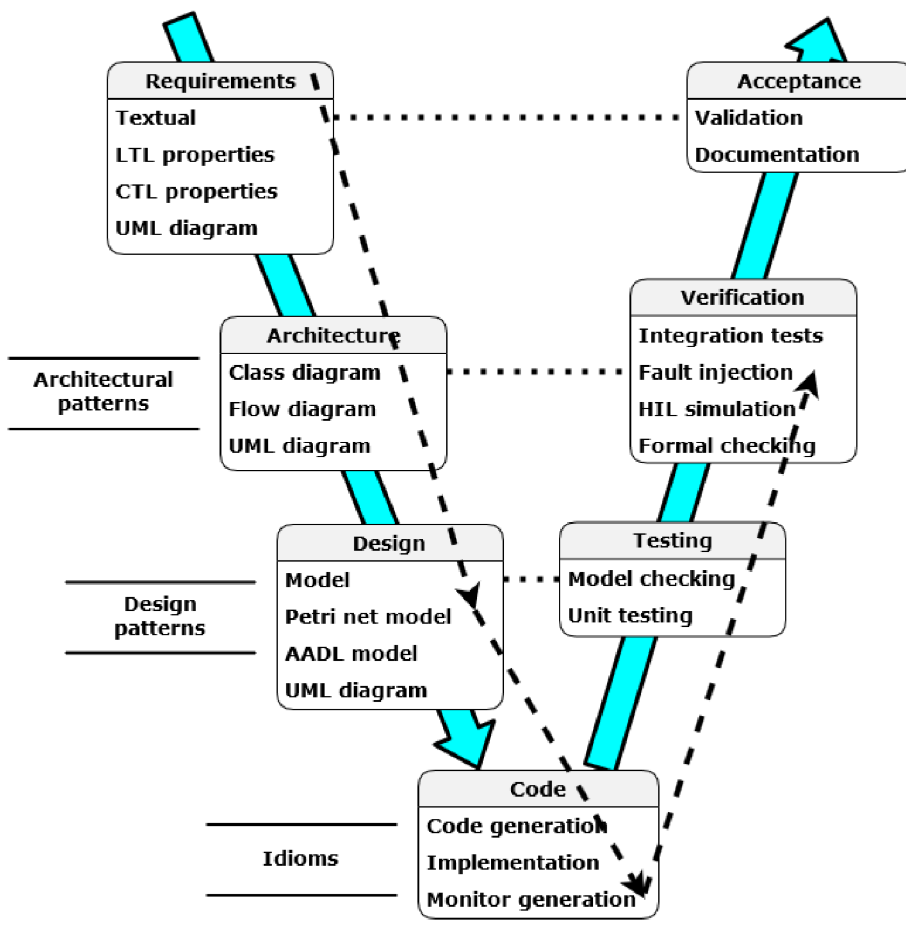
A PŘÍLOHY

A.1 Implementace uzlu (místo + hrana) kontrolního modelu ve VHDL

```
entity PetriNode is
    generic (
        pc_start : natural := 5;
        pc_end : natural := 10;
        earliest : natural := 10;
        latest : natural := 20
    );
    port (
        clock: in std_logic;
        pc: in std_logic;
        input: in std_logic;
        output: out std_logic;
        timeout: out std_logic;
        error: out std_logic
    );
end PetriNode;

architecture petri_node_behave of PetriNode is
    signal r_counter : natural range 0 to 1000;
    signal r_marking : std_logic = '0';
    signal r_output: std_logic = '0';
    signal r_timeout: std_logic = '0';
    signal r_error: std_logic = '0';
begin
    monitor : process (clock) is
    begin
        if rising_edge(clock) then
            if input = '1' then
                r_marking = '1';
                t_timeout <= '0';
                r_output <='0';
                r_error <= '0';
            end if;

            if r_marking = '0' then
                if (pc >= pc_start and pc <= pc_end) then
                    r_error <= '1';
                    r_timeout <= '0';
                    r_output <='0';
                else
                    r_error <= '0';
                    r_timeout <= '0';
                    r_output <='0';
                end if;
            else
                if pc = pc_end then
                    if (r_counter < earlist or r_counter > latest) then
                        t_timeout <= '1';
                        r_output <='0';
                        r_error <= '0';
                    else
                        t_timeout <= '0';
                        r_output <='1';
                        r_error <= '0';
                    end_if;
                else
                    t_timeout <= '0';
                    r_output <='0';
                    r_error <= '0';
                    if (pc >= pc_start and pc <= pc_end) then
                        r_counter <= r_counter + 1;
                    end
                end if;
            end if;

            error <= r_error;
            output <= r_output;
            timeout <= r_timeout;
        end if;
    end process monitor;
end petri_node_behave;
```

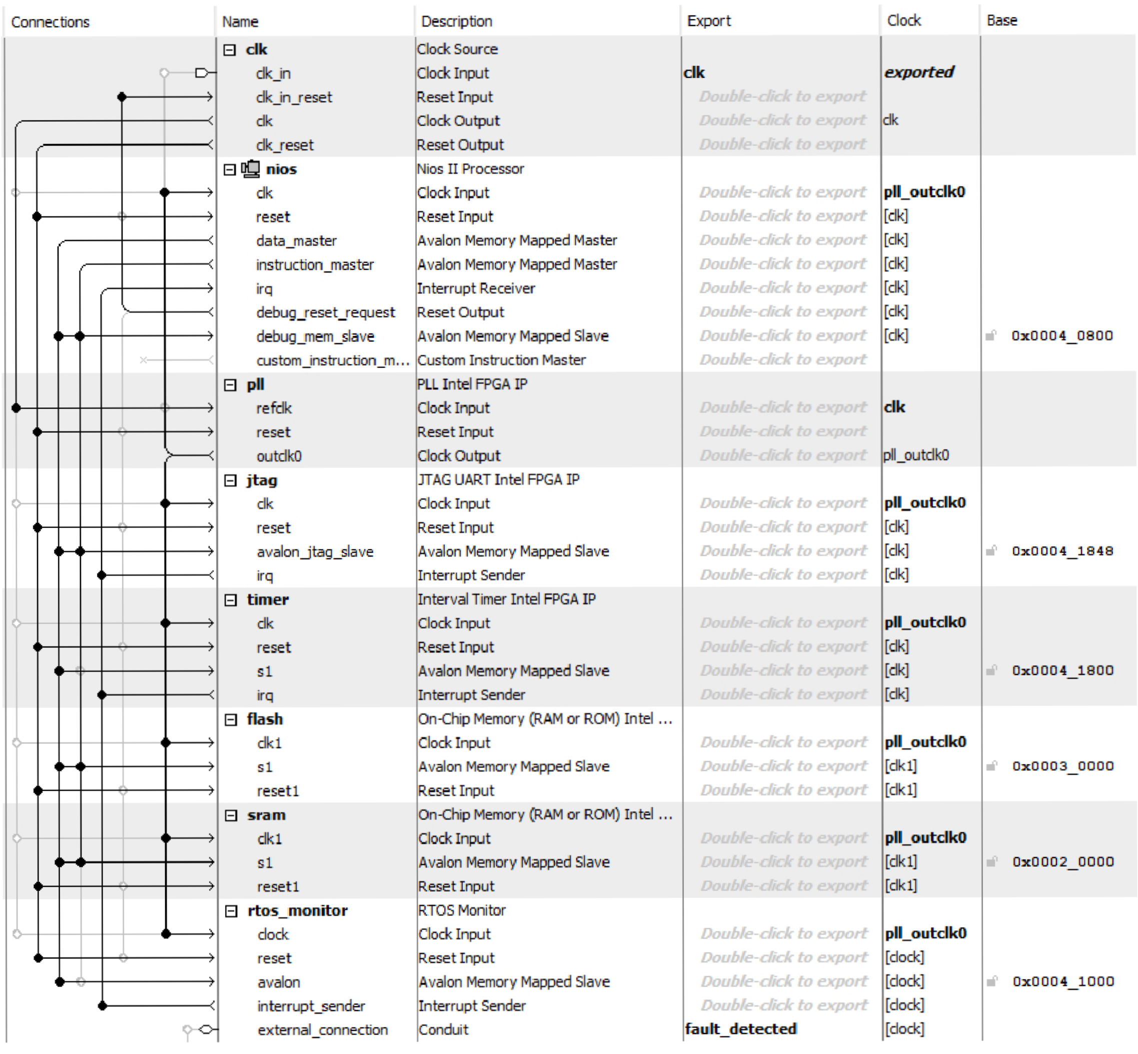
Obr. A.1: Implementace uzlu (místo + hrana) kontrolního modelu ve VHDL

A.2 Integrace procesu vytvoření kontrolního systému do standardního V-modelu



Obr. A.2: Integrace procesu vytvoření kontrolního systému do standardního V-modelu

A.3 Návrh hardware platformy pro měření událostí pomocí mapované periferie



Connections	Name	Description	Export	Clock	Base
	⊟ clk	Clock Source			
	clk_in	Clock Input	clk	*exported*	
	clk_in_reset	Reset Input	*Double-click to export*		
	clk	Clock Output	*Double-click to export*	clk	
	clk_reset	Reset Output	*Double-click to export*		
	⊟ nios	Nios II Processor			
	clk	Clock Input	*Double-click to export*	**pll_outclk0**	
	reset	Reset Input	*Double-click to export*	[clk]	
	data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]	
	instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]	
	irq	Interrupt Receiver	*Double-click to export*	[clk]	
	debug_reset_request	Reset Output	*Double-click to export*	[clk]	
	debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_0800
	custom_instruction_m...	Custom Instruction Master	*Double-click to export*		
	⊟ pll	PLL Intel FPGA IP			
	refclk	Clock Input	*Double-click to export*	**clk**	
	reset	Reset Input	*Double-click to export*		
	outclk0	Clock Output	*Double-click to export*	pll_outclk0	
	⊟ jtag	JTAG UART Intel FPGA IP			
	clk	Clock Input	*Double-click to export*	**pll_outclk0**	
	reset	Reset Input	*Double-click to export*	[clk]	
	avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1848
	irq	Interrupt Sender	*Double-click to export*	[clk]	
	⊟ timer	Interval Timer Intel FPGA IP			
	clk	Clock Input	*Double-click to export*	**pll_outclk0**	
	reset	Reset Input	*Double-click to export*	[clk]	
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0004_1800
	irq	Interrupt Sender	*Double-click to export*	[clk]	
	⊟ flash	On-Chip Memory (RAM or ROM) Intel ...			
	clk1	Clock Input	*Double-click to export*	**pll_outclk0**	
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0003_0000
	reset1	Reset Input	*Double-click to export*	[clk1]	
	⊟ sram	On-Chip Memory (RAM or ROM) Intel ...			
	clk1	Clock Input	*Double-click to export*	**pll_outclk0**	
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0002_0000
	reset1	Reset Input	*Double-click to export*	[clk1]	
	⊟ rtos_monitor	RTOS Monitor			
	clock	Clock Input	*Double-click to export*	**pll_outclk0**	
	reset	Reset Input	*Double-click to export*	[clock]	
	avalon	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0004_1000
	interrupt_sender	Interrupt Sender	*Double-click to export*	[clock]	
	external_connection	Conduit	**fault_detected**	[clock]	

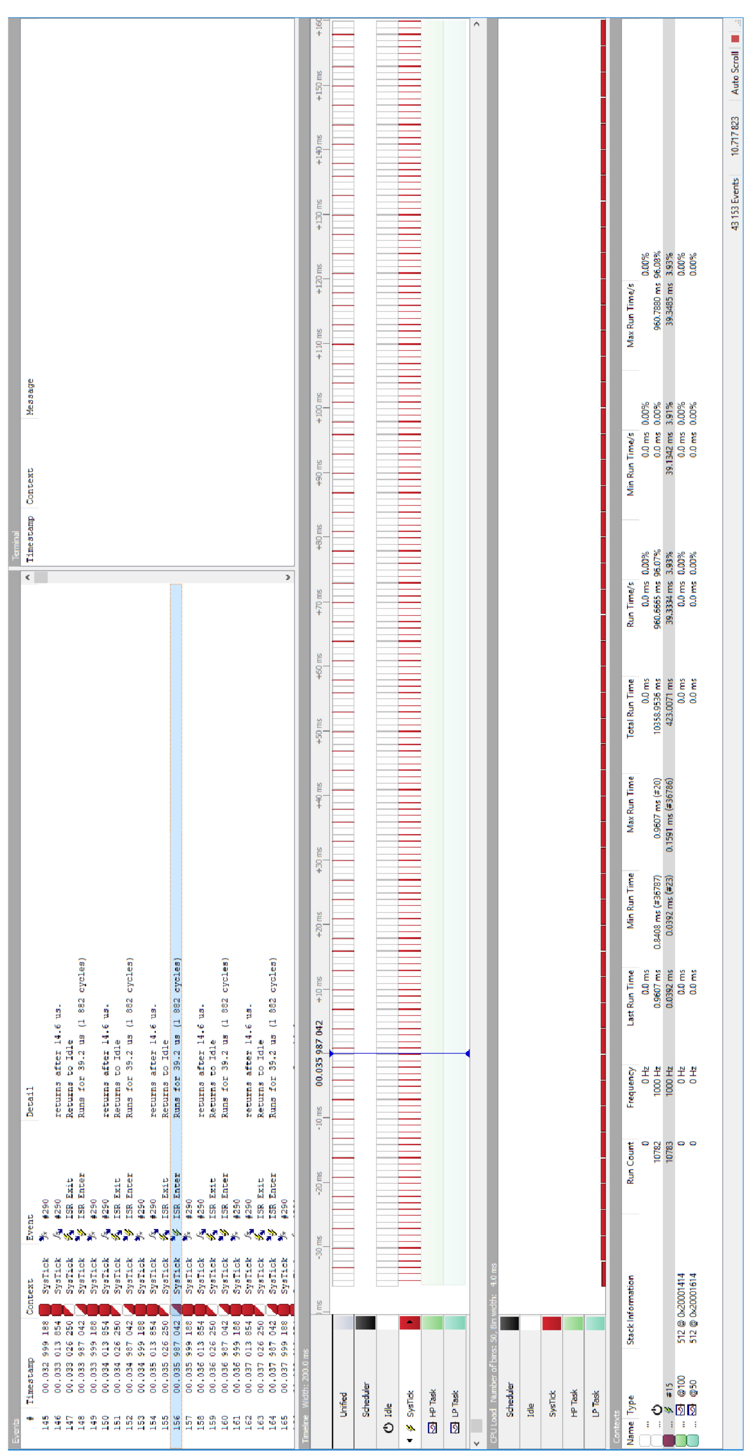
Obr. A.3: Návrh hardware platformy pro měření událostí pomocí mapované periferie

A.4 Implementace scénáře 1 – *deadlock*

```
/*
 * high priority task function
 */
static void HPTask(void) {
  while (1) {
18    if (OS_MUTEX_LockBlocked(&M1))
    {
20      OS_TASK_Delay(5);
      if (OS_MUTEX_LockBlocked(&M2))
      {
        BSP_ToggleLED(0);
        OS_MUTEX_Unlock(&M2);
      }
      OS_MUTEX_Unlock(&M1);
    }
    OS_TASK_Delay(50);
  }
30 }

/*
 * low priority task function
 */
static void LPTask(void) {
  while (1) {
    if (OS_MUTEX_LockBlocked(&M2))
    {
      if (OS_MUTEX_LockBlocked(&M1))
40      {
        BSP_ToggleLED(1);
        OS_MUTEX_Unlock(&M1);
      }
      OS_MUTEX_Unlock(&M2);
    }
    OS_TASK_Delay(50);
  }
}
```

Obr. A.4: Implementace scénáře 1 – *deadlock*

A.5 Záznam událostí RTOS ze scénáře 1 – *deadlock*



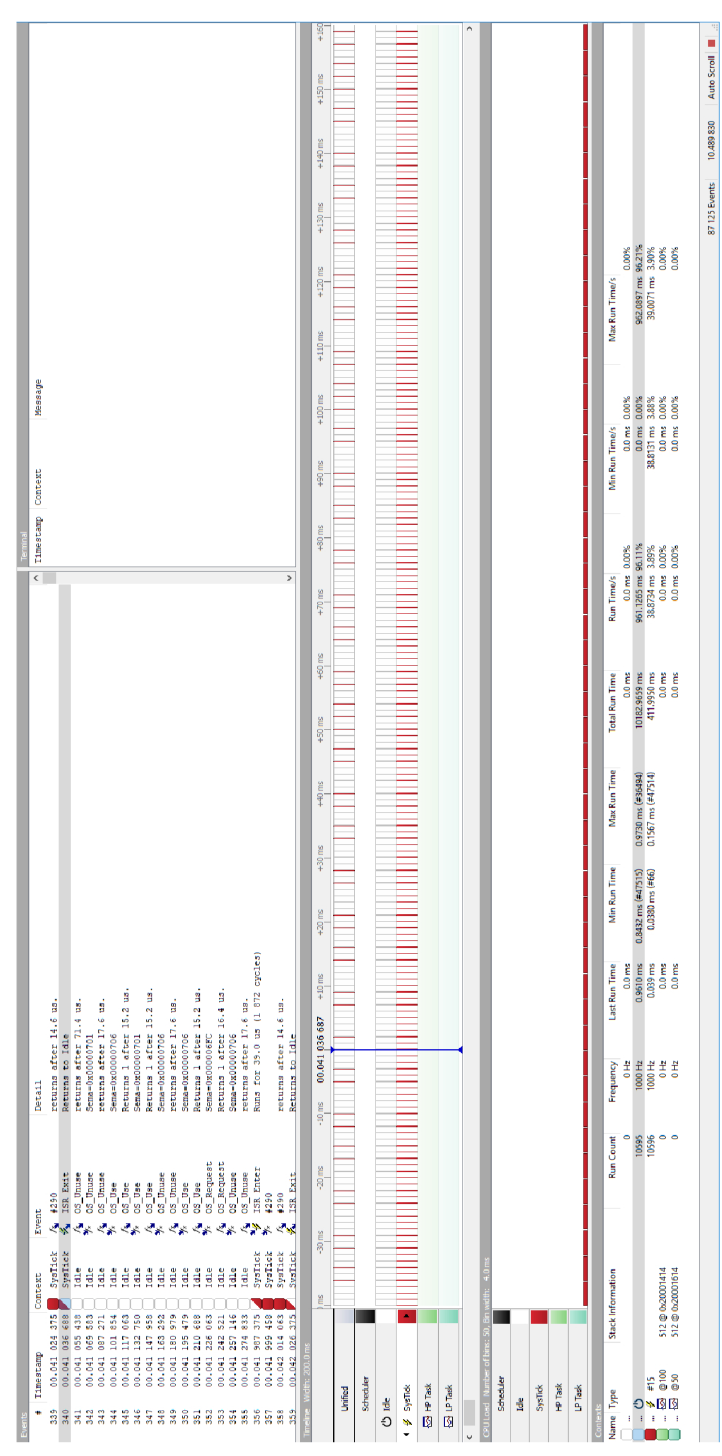
Obr. A.5: Záznam událostí RTOS ze scénáře 1 – *deadlock*

A.6 Implementace scénáře 2 – *livelock*

```c
/*
 * high priority task function
 */
static void HPTask(void) {
  char zamek = 0;
  while (1) {
    OS_MUTEX_LockBlocked(&MG);
    OS_MUTEX_LockBlocked(&M1);
    OS_MUTEX_Unlock(&MG);

    OS_MUTEX_LockBlocked(&MG);
    zamek = OS_MUTEX_Lock(&M2);
    OS_MUTEX_Unlock(&MG);
    if (zamek == 0)
    {
      OS_MUTEX_Unlock(&M1);
      continue;
    }
    BSP_ToggleLED(0);
    for(int i = 0; i < 10000; i++) {}
    OS_MUTEX_Unlock(&M2);
    OS_MUTEX_Unlock(&M1);
  }
}

/*
 * low priority task function
 */
static void LPTask(void) {
  char zamek = 0;
  while (1) {
    OS_MUTEX_LockBlocked(&MG);
    OS_MUTEX_LockBlocked(&M2);
    OS_MUTEX_Unlock(&MG);

    OS_MUTEX_LockBlocked(&MG);
    zamek = OS_MUTEX_Lock(&M1);
    OS_MUTEX_Unlock(&MG);
    if (zamek == 0)
    {
      OS_MUTEX_Unlock(&M2);
      continue;
    }
    BSP_ToggleLED(1);
    for(int i = 0; i < 10000; i++) {}
    OS_MUTEX_Unlock(&M1);
    OS_MUTEX_Unlock(&M2);
  }
}
```

Obr. A.6: Implementace scénáře 2 – *livelock*

A.7 Záznam událostí RTOS ze scénáře 2 – *livelock*



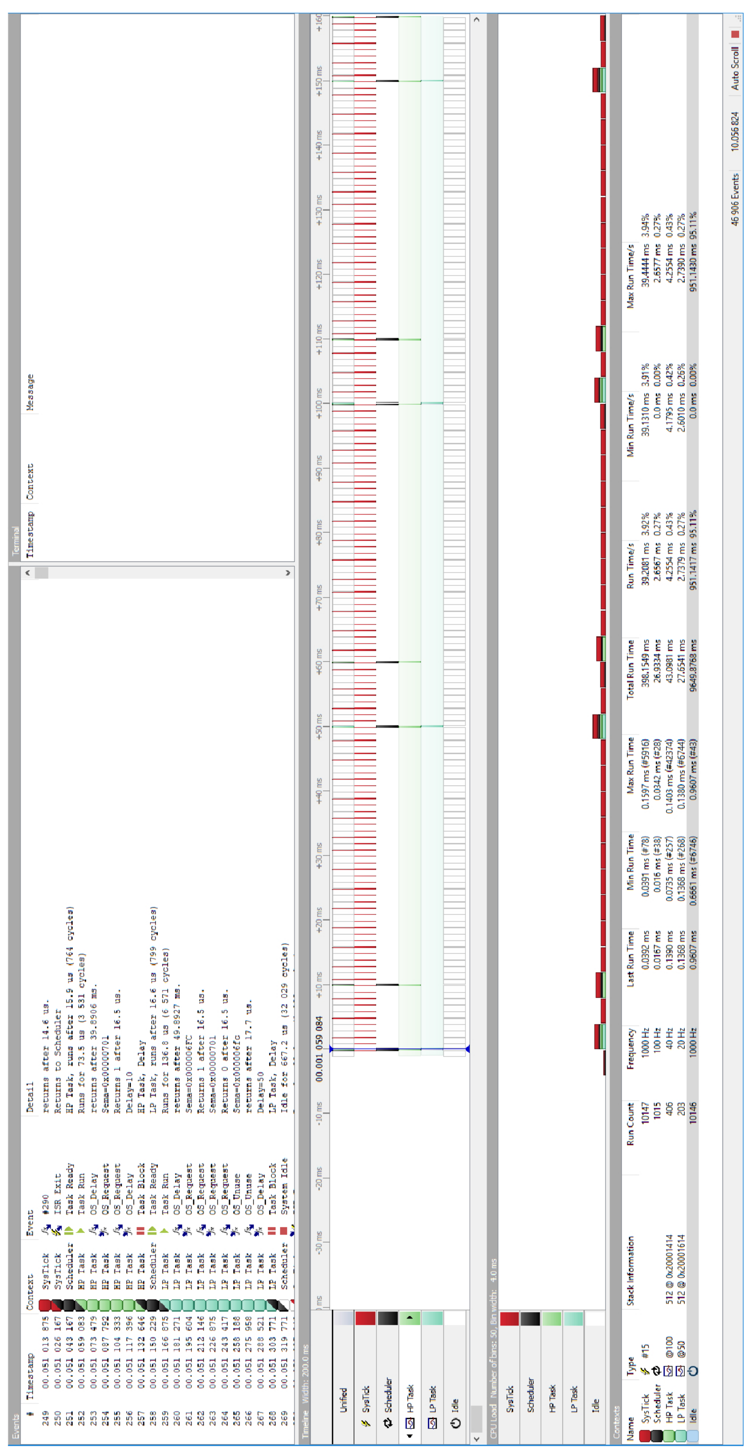
Obr. A.7: Záznam událostí RTOS ze scénáře 2 – *livelock*

A.8 Implementace scénáře 3 – *starvation*

```
/*
 * high priority task function
 */
static void HPTask(void) {
  while (1) {
    if (OS_MUTEX_Lock(&M1))
    {
      OS_TASK_Delay(10);
      if (OS_MUTEX_Lock(&M2))
      {
        BSP_ToggleLED(0);
        OS_MUTEX_Unlock(&M2);
      }
      OS_MUTEX_Unlock(&M1);
    }
    OS_TASK_Delay(40);
  }
}

/*
 * low priority task function
 */
static void LPTask(void) {
  while (1) {
    if (OS_MUTEX_Lock(&M2))
    {
      if (OS_MUTEX_Lock(&M1))
      {
        BSP_ToggleLED(1);
        OS_MUTEX_Unlock(&M1);
      }
      OS_MUTEX_Unlock(&M2);
    }
    OS_TASK_Delay(50);
  }
}
```

Obr. A.8: Implementace scénáře 3 – *starvation*

A.9 Záznam událostí RTOS ze scénáře 3 – *starvation*



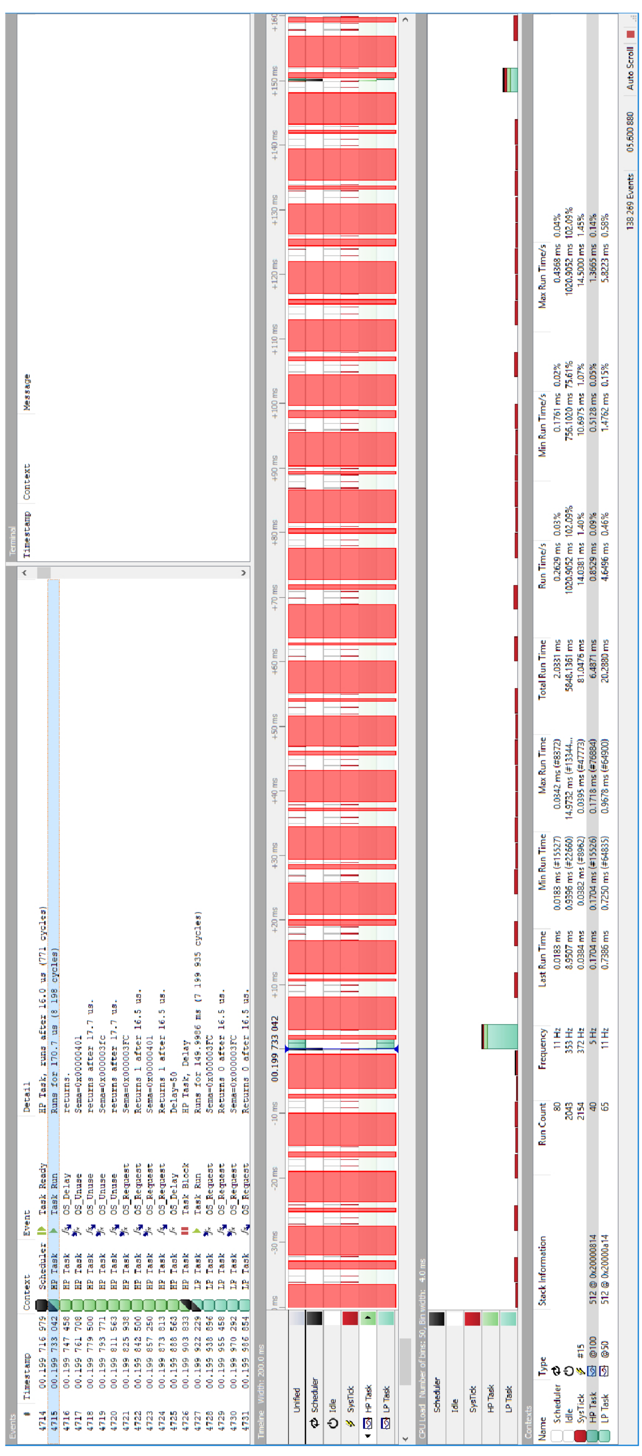
Obr. A.9: Záznam událostí RTOS ze scénáře 3 – *starvation*

A.10 Implementace scénáře 4 – *race condition*

```
/*
 * high priority task function
 */
static void HPTask(void) {
  while (1) {
    if (OS_MUTEX_Lock(&M1))
    {
20    if (OS_MUTEX_Lock(&M2))
      {
        BSP_ToggleLED(0);
        OS_TASK_Delay(50);
        OS_MUTEX_Unlock(&M2);
      }
      OS_MUTEX_Unlock(&M1);
    }
  }
}
30
/*
 * low priority task function
 */
static void LPTask(void) {
  while (1) {
    if (OS_MUTEX_Lock(&M1))
    {
      if (OS_MUTEX_Lock(&M2))
      {
40      BSP_ToggleLED(1);
        OS_TASK_Delay(100);
        OS_MUTEX_Unlock(&M2);
      }
      OS_MUTEX_Unlock(&M1);
    }
  }
}
```

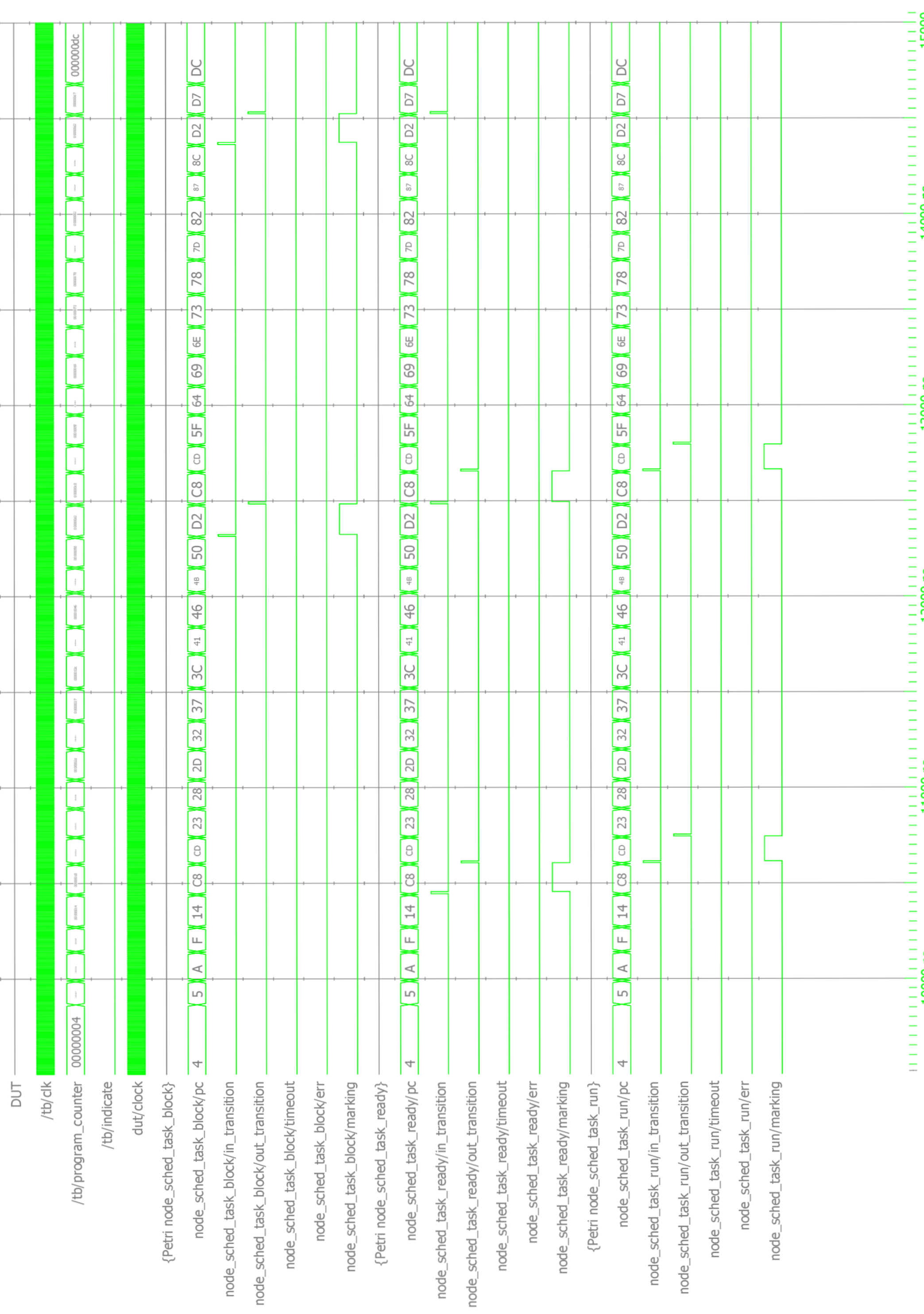
Obr. A.10: Implementace scénáře 4 – *race condition*

A.11 Záznam událostí RTOS ze scénáře 4 – *race condition*



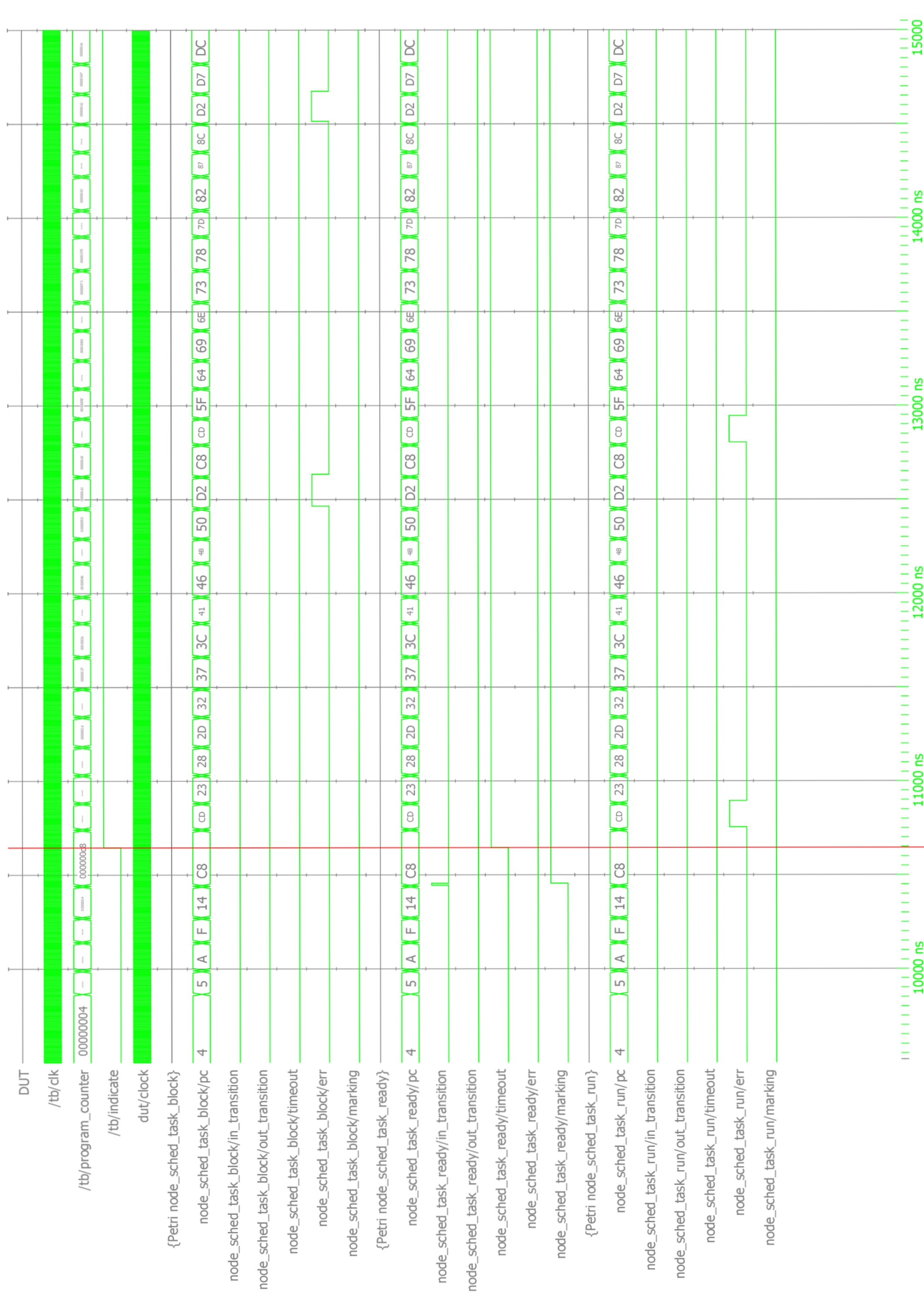
Obr. A.11: Záznam událostí RTOS ze scénáře 4 – *race condition*

A.12 Záznam událostí simulace třídy I – normální běh



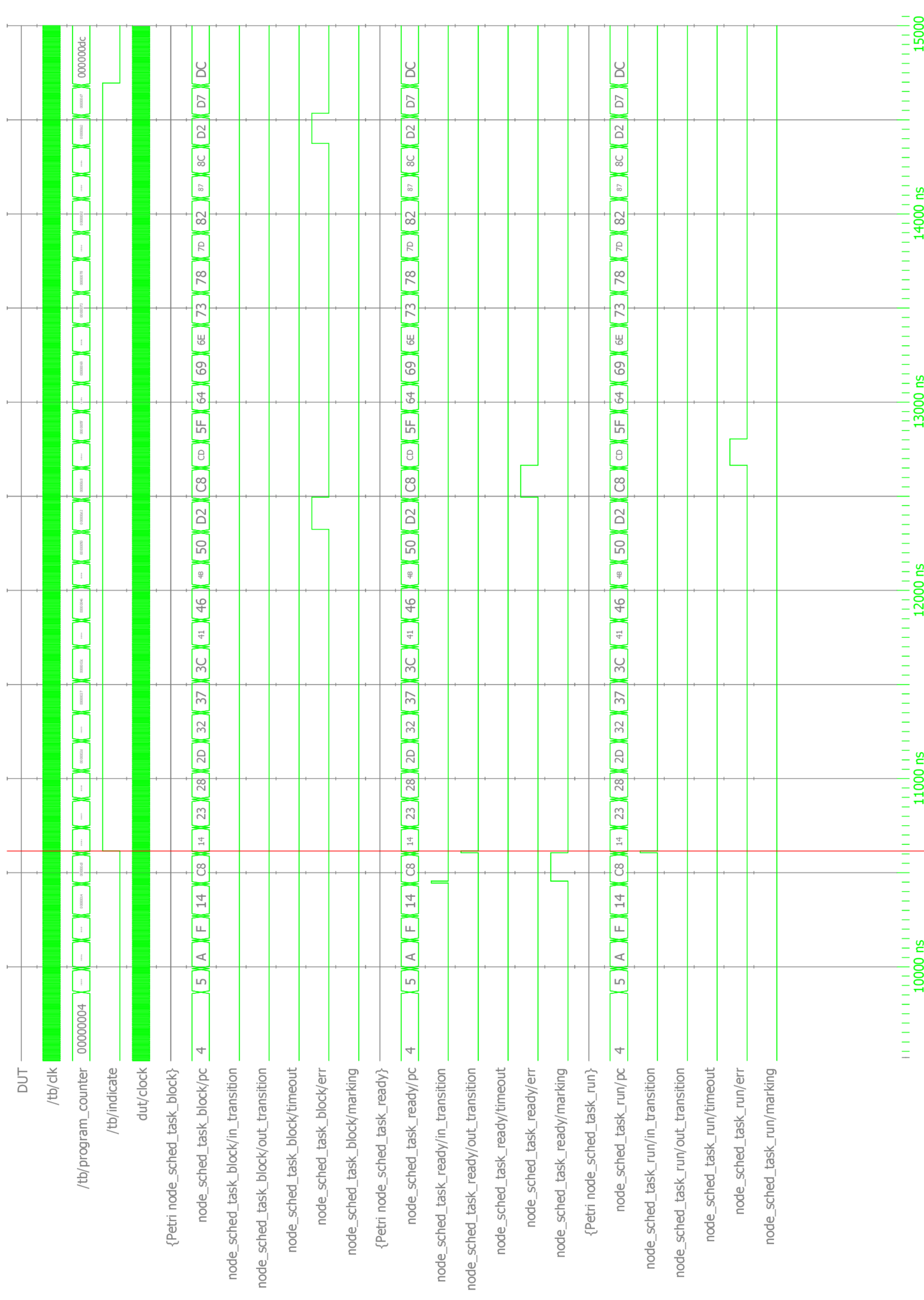
Obr. A.12: Průběh vybraných signálů simulace třídy I – **normální běh**

A.13 Záznam událostí simulace třídy II – časová chyba



Obr. A.13: Průběh vybraných signálů simulace třídy II – **časová chyba**

A.14 Záznam událostí simulace třídy III – chyba kontinuity



Obr. A.14: Průběh vybraných signálů simulace třídy III – **chyba kontinuity**