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 - ZKRÁCENÁ VERZE
DOCTORAL THESIS - SHORTEND VERSION

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 - zkrácená verze. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky, 2020. 27 s. Vedoucí práce byl doc. Ing. Zdeněk Bradáč, Ph.D.

OBSAH

1 ÚVOD

Důležitou společnou vlastností vestavných 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.

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*).

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.

2 FUNKČNÍ BEZPEČNOST RTOS

V této kapitole popíši relevantní teoretické základy a také inovativní přístupy v oblasti funkční bezepčnosti vztažené k operačním systémům reálného času, na kterých budu stavět v dalších kapitolách. Cílem není popsat vyčerpávajícím způsobem současné informace a znalosti, ale uchopit současný stav poznání relevantně k tématu práce.

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.

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*)." [8] 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 [10].

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).

Každá úloha má své parametry, 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.

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.

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.

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.

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, 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.

Chyby v RTOS aplikaci propagují často z hardwarových defektů (*soft-errors*). [7] 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 [6]. 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).

Obecně lze zdroje chyb v doméně vestavných systémů, resp. integrovaných obvodů dělit 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

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.

2.3 Návrhové vzory

Návrhový vzor v programátorské praxi je definován jako *obecné řešení častého problému při návrhu software* [2] 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.

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.

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.

2.4 Inovativní přístupy

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.

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. 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 $99,2$ %.

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 [5]. 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 periferií.

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 [11].

V [12] 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.

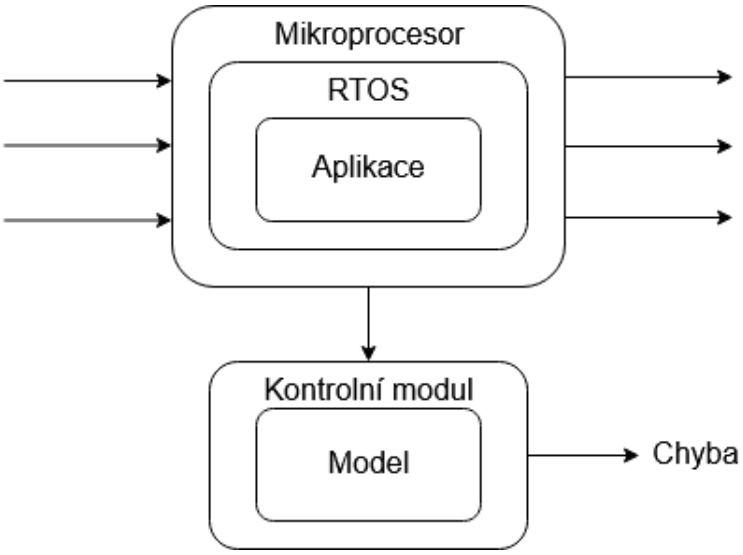
V [9] jsou pro detekci chyb využity výpočty řešitelnosti plánování (angl *feasibility*) v závislosti na běžících úlohách a jejich nejhorších časech. Tato softwarová kontrola se provádí v rámci úkonu přepnutí kontextu. Metoda odhalí chyby, které způsobí zmeškání deadline a jiné chyby meziprocesové komunikace, avšak spočítat funkci užitečnosti (angl. *utility function*) je pro některé algoritmy (např. RMS) obtížné, ne-li pouze orientační.

[1] 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.

[1] 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.

3 NOVÁ NAVRHOVANÁ METODA

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 3.1.

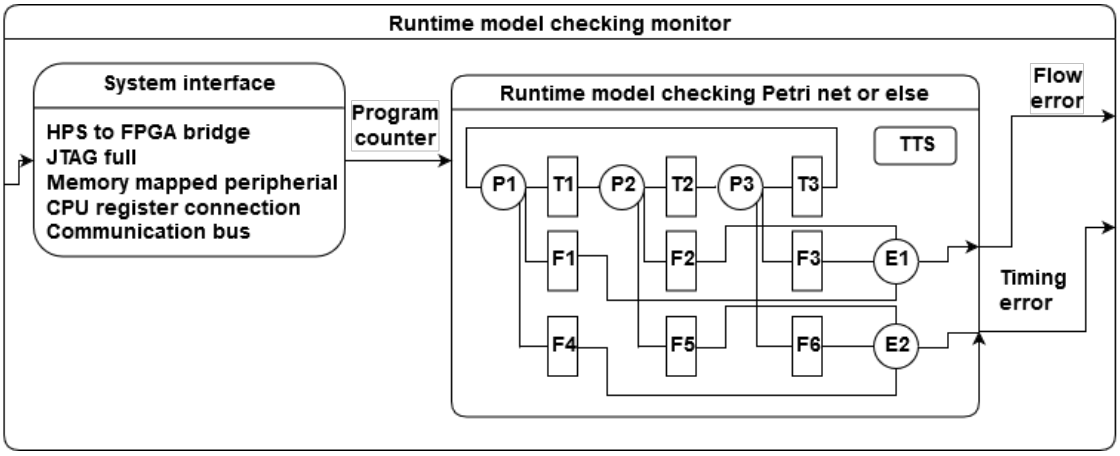


Obr. 3.1: Architektura systému online kontroly systému

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. 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 [11]). 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.

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. 3.2.



Obr. 3.2: Blokové schéma online kontrolního systému

3.1 Model

Specifikace modelu sledovaného systému určuje obor chyb, který je možné daným online kontrolním systémem detekovat. Mimo přístupy uvedené v kap. 2.4, jimiž jsou monitorování toku programu (viz [3]) 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.

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.2 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.

3.3 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;
- $\bullet e : P \times T \to \mathbb{N}$ je zpětná přechodová funkce;
- $e\bullet : P \times T \to \mathbb{N}$ 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 $\Gamma \in [\Gamma_\alpha(t), \Gamma_\beta(t)]$;

- $\Psi : T \rightarrow \mathbb{N}^+$ 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:

- $M : P \rightarrow \mathbb{N}$ 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 [4]). 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 $TTS = (S, S_0, A, C, I, \rightarrow)$, kde:

- S je množina stavů, přičemž počáteční stav $S_0 \subseteq S$;
- A je množina akcí;
- C je konečná množina proměnných obsahujících časovou značku;
- $I : S \rightarrow \Phi(C)$ přiřazuje časovou značku stavu;
- $\rightarrow \in S \times T \times 2^C \times S$ je binární relace nad množinou stavů, označovaná jako

přechod.

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

$$(M,\nu) \xrightarrow{t_i} (M',\nu') iff \begin{cases} PC = \Psi(t_i) \wedge \nu_i \in I(t_i) \\ M = pre(t_i) \end{cases} \tag{3.1}$$

$$(M,\nu) \xrightarrow{\epsilon_d} (M,\nu') iff \begin{cases} \nu_i' = \begin{cases} \nu_i + d & \text{if} PC \in \Gamma(p_i) \\ \nu_i' & \text{jinak.} \end{cases} \\ M \geq pre(t_i) \end{cases} \tag{3.2}$$

$$F' = F + \phi(t_i) iff \begin{cases} PC = \Psi(t_i) \wedge \nu_i \notin I(t_i) \\ PC \in \Gamma(p_i) \wedge M \neq pre(t_i) \\ (M,\nu) \xrightarrow{t_i} (M',\nu') \wedge M \geq pre(t_i) \end{cases} \tag{3.3}$$

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.

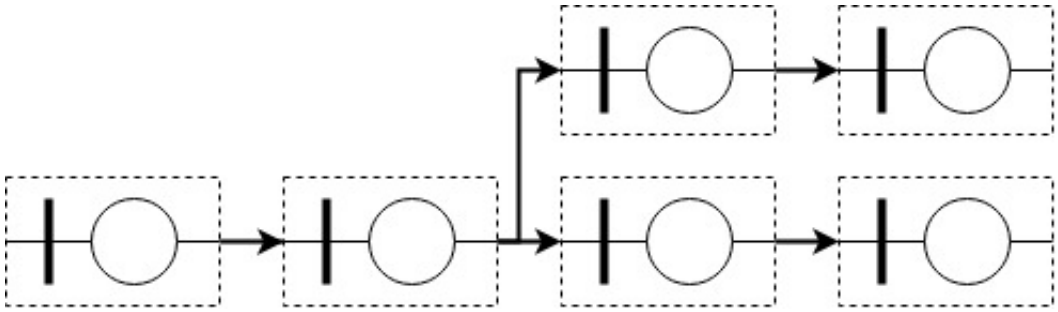
3.4 Implementace

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. 3.3). Tyto uzly je možné seskupovat za účelem vytvoření programových linií a také větvit (viz příklad sítě na obr. 3.4).

Obr. 3.3: Diagram uzlu

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

- manuálním vytvořením propojených uzlů dle předpisu modelu,
- generováním VHDL kódu s následným doplněním adres z vygenerovaného mapovacího souboru vzniklého při kompilaci aplikace.



Obr. 3.4: Příklad kontrolní Petriho sítě skládající se z uzlů

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*).

3.5 Definice vzoru architektury

V této podkapitole se pokusím o definici vzoru architektury, jehož komponenty jsem doposud rozebíral. 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.

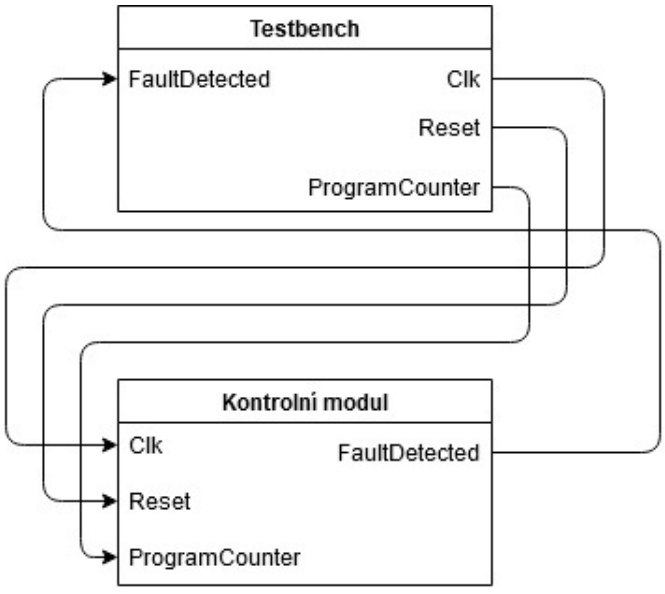
Důsledky (angl. *consequences*): Mezi přínosy patří, že tato technika eliminuje chyby se společnou příčinou. Přínosem také je minimální zatížení monitorované procesorové jednotky v závislosti na stupni integrace (napojení na sběrnici procesoru nebo jako mapovaná periferie). Nevýhodou je, že musíte vytvořit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

4 SIMULACE

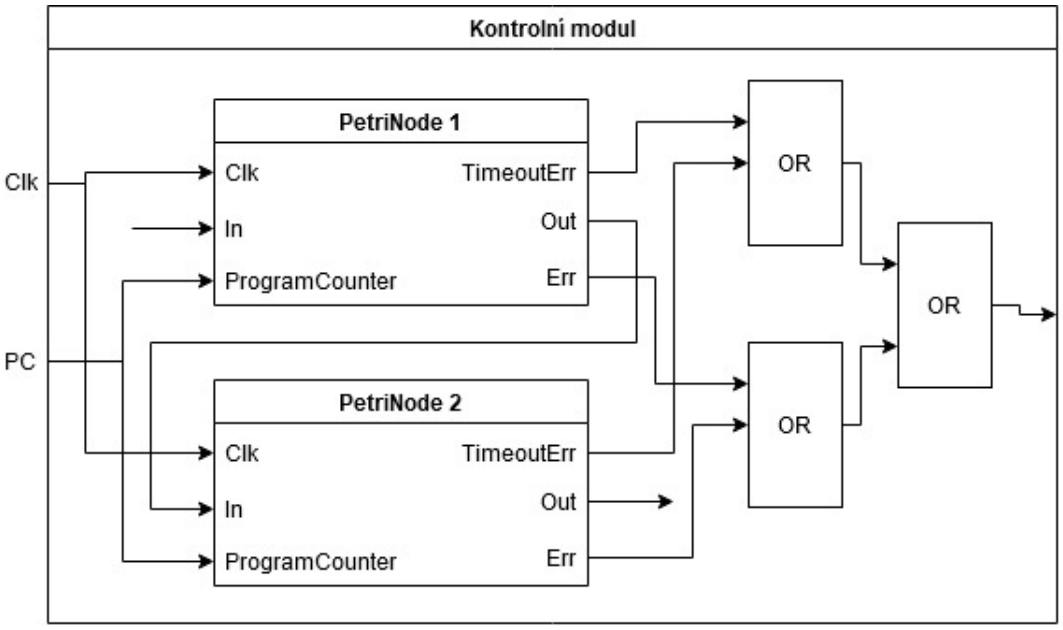
Jelikož je reálná implementace a integrace navrhována pro prostředí hradlového pole, implementoval jsem proto kontrolní subsystém pomocí nástroje **Quartus** a **ModelSim**, což jsou nástroje pro programování a simulaci hradlových polí firmy **Altera**. Jako programovací jazyk pro implementaci kontrolního 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.

Rozhodl jsem se 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 4.1. Blokové schéma samotného kontrolního modulu (viz obr. 4.2) ukazuje, že základním vyhodnocovacím modulem je *PetriNode*, který implementuje místo a přechod jakožto jeden uzel **kontrolní Petriho sítě**. Definici modulu *PetriNode* zachycuje obr. 4.3.



Obr. 4.1: Blokové schéma testovací architektury



Obr. 4.2: Blokové schéma kontrolního modulu

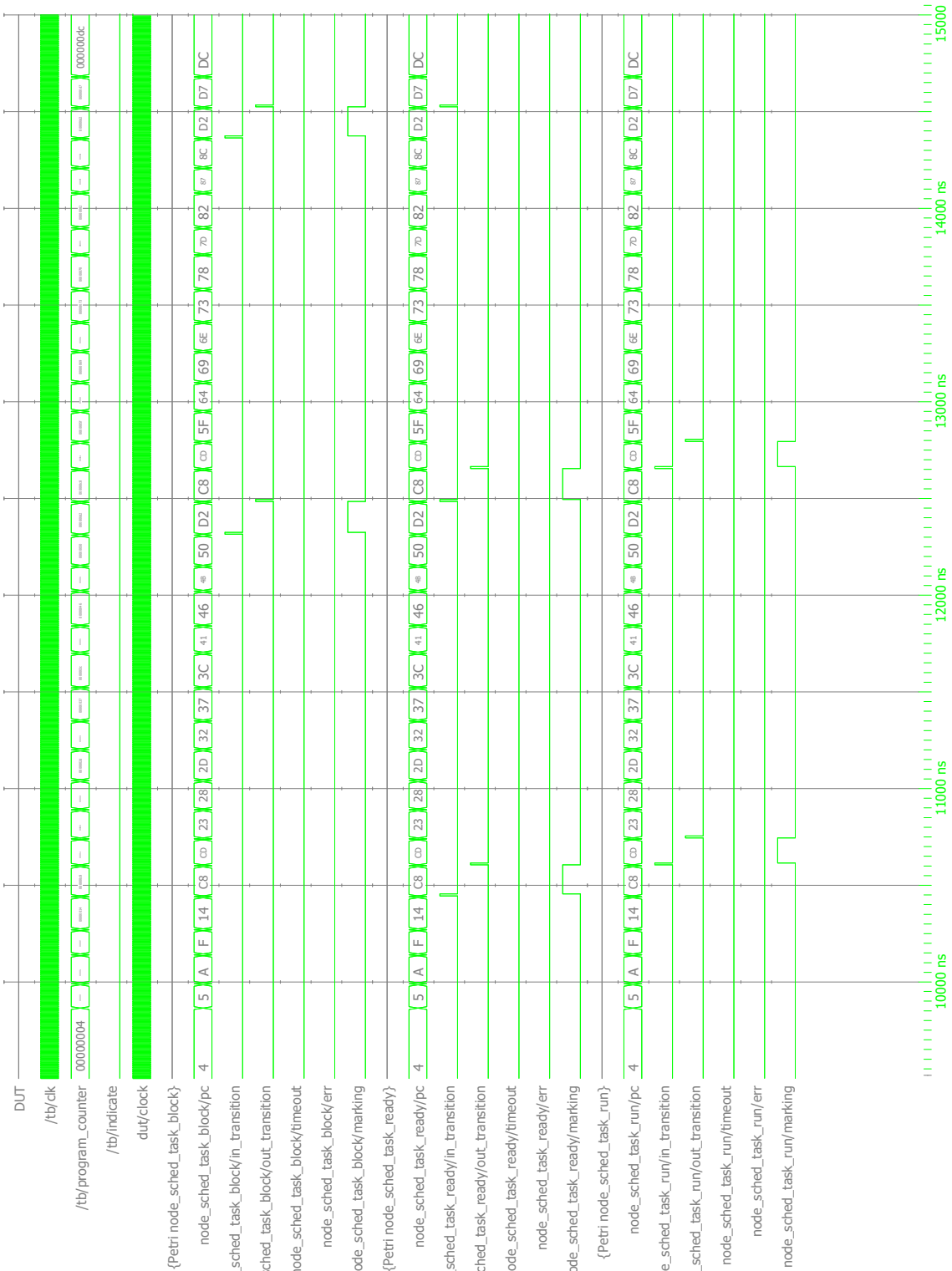
```
entity petri_node is
    generic (
        pc_start : natural := 5;
        pc_end : natural := 10;
        earliest : natural := 10;
        latest : natural := 20
    );
    port (
        clk: in std_logic;
        pc: in natural;
        in_transition: in std_logic;
        out_transition: out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

Obr. 4.3: Definice modulu *PetriNode* v jazyce VHDL

4.1 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 na obrázku 4.4.



Obr. 4.4: Průběh vybraných signálů simulace třídy I – **normální běh**

4.2 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 na obrázku 4.5.

4.3 Třída III – chyba kontinuity

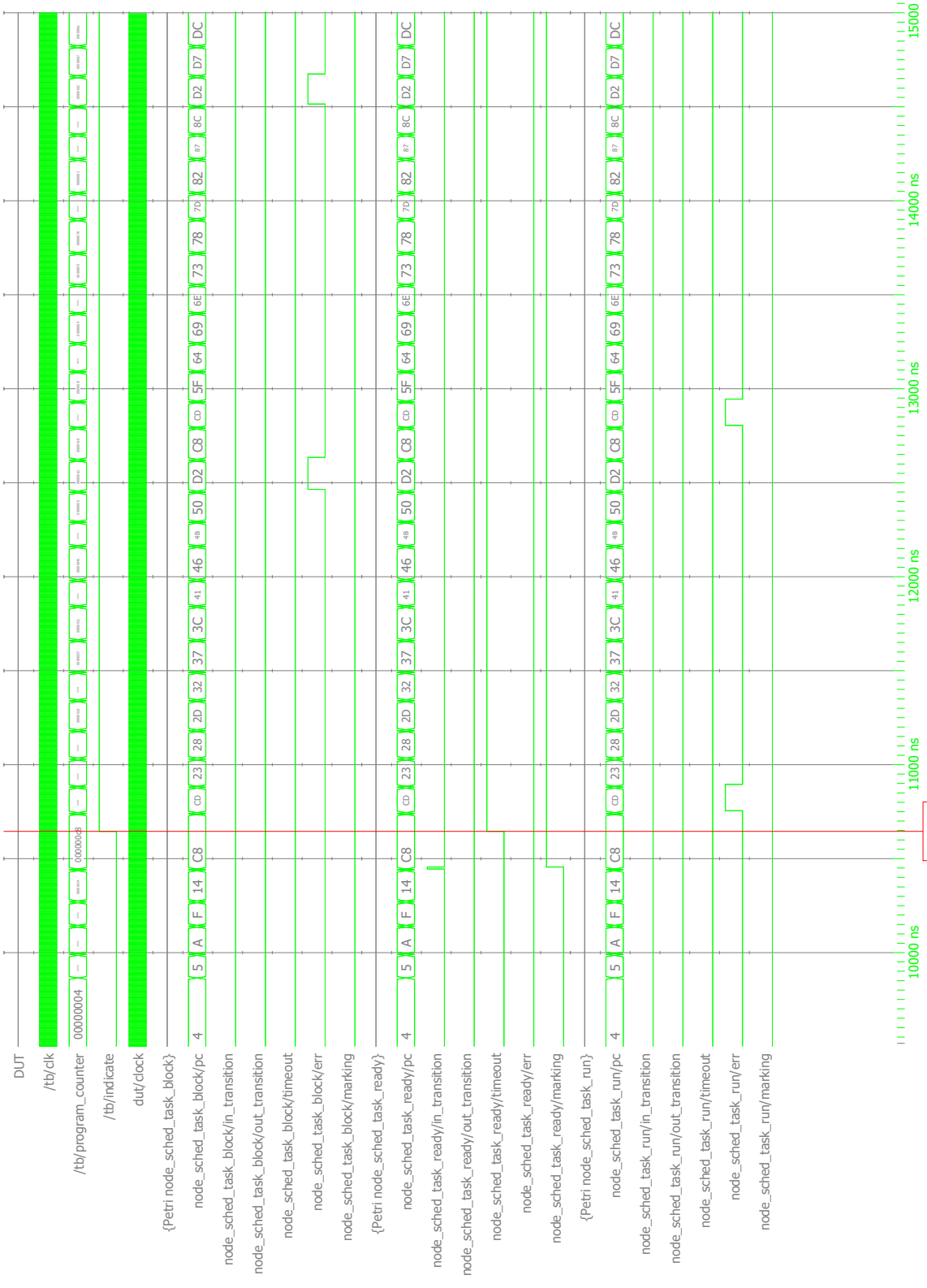
Jedná se o situace, kdy nastala chyba porušení pravidel kontinuity (sledu událostí) dle modelu. Je tedy vnesena chyba v podobě zápisu jiné hodnoty registru *Program Counter*, než v dané situaci může být. Tato situace odpovídá případu, kdy se vlivem defektu (nebo cíleného útoku) přepíše registr *Program Counter* a začne se vykonávat nežádoucí (jiný) úsek kódu. Kontrolní modul tedy nastaví svůj výstup *FaultDetected*, čímž signalizuje chybu. Časový záznam se nachází na obrázku 4.6.

4.4 Diskuze výsledků

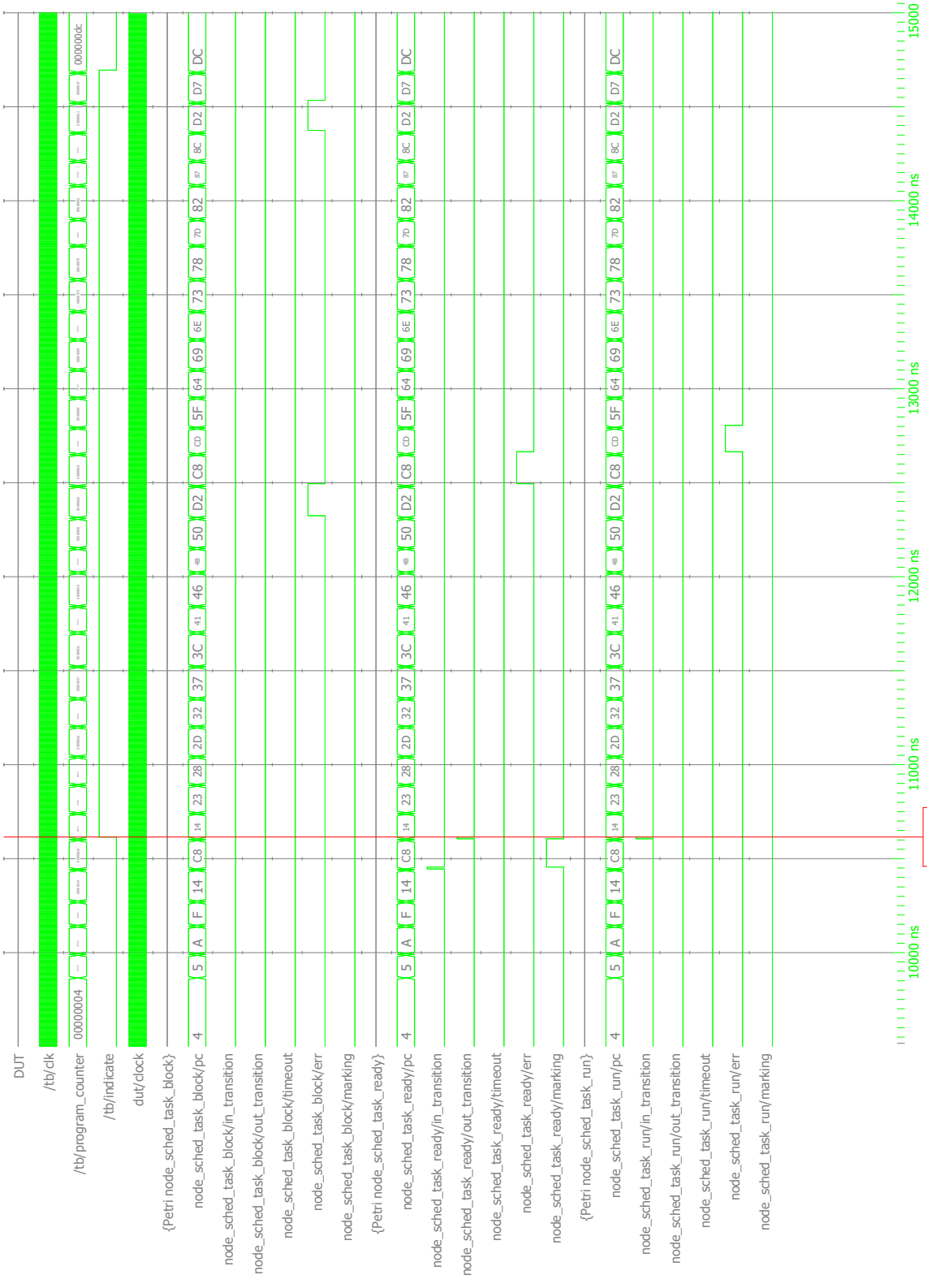
Dle výsledků simulace uvedených v kap. 4 mohu konstatovat, že kontrolní modul úspěšně detekoval chyby v testovací sadě. Abych mohl deklarovat plnou funkčnost, musí být ještě provedena verifikace, což je komplexní činnost přesahující rámec této práce. Z programátorského hlediska ale mohu prohlásit, že 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*.

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).



Obr. 4.5: Průběh vybraných signálů simulace třídy II – **časová chyba**



Obr. 4.6: Průběh vybraných signálů simulace třídy III – **chyba kontinuity**

5 BUDOUCÍ PRÁCE

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.

6 ZÁVĚR

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*.

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

- 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.

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.

LITERATURA

[1] 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.

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

[3] 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

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] 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/

[11] 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.

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

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.; 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.
- 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.