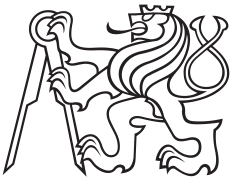
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra řídící techniky

Modifikace nástroje TaSysTest pro běh v prostředí Real-Time

Žák Pavel Bc.
Počítačové inženýrství, Otevřená informatika
zakpave3@fel.cvut.cz

Červen 2016
Vedoucí práce: Sobotka Jan Ing.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Pavel Žák**

Studijní program: Otevřená informatika
Obor: Počítačové inženýrství

Název tématu: **Modifikace nástroje TaSysTest pro běh v prostředí Real-Time**

Pokyny pro vypracování:

1. Seznamte se s testovacím nástrojem TaSysTest [1].
2. Navrhněte vhodné úpravy architektury software pro portování do prostředí reálného času LabView Real-Time.
3. Software upravte pro využití v prostředí LabView Real-Time a komunikace s NI VeriStand.
4. Navržené úpravy implementujte.
5. Demonstrujte funkčnost provedené implementace na NI PXIe-8135.

Seznam odborné literatury:

[1] GRUS, Tomáš. Implementace softwarového nástroje pro generování integračních testů. Praha, 2014. Diplomová práce
[2] J. Zander, I. Schieferdecker, and P.J. Mosterman. Model-Based Testing for Embedded Systems. Taylor & Francis, 2011
[3] CONWAY, Jon a Steve WATTS. A software engineering approach to LabVIEW. Upper Saddle River, NJ: Prentice Hall, Professional Technical Reference, c2003, xiii, 221 p. ISBN 01-300-9365-3

Vedoucí: Ing. Jan Sobotka

Platnost zadání: do konce letního semestru 2016/2017

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 15. 10. 2015

Poděkování / Prohlášení

Na tomto místě bych rád poděkoval vedoucímu práce panu Ing. Janu Sobotkovi za odborné vedení, za pomoc, za cenné odborné rady a připomínky a v neposlední řadě za ochotu a trpělivost při vypracovávání této práce.

Také bych chtěl poděkovat mým rodičům za podporu, jak morální tak hmotnou a i všem mým blízkým za oporu a motivaci po celou dobu a hlavně na konci studia.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 27.5.2016

.....

Abstrakt / Abstract

Cílem této práce je rozšíření nástroje TASysTest [1] o možnost běhu jeho výpočetního jádra v prostředí Real-Time při dodržení jeho původní koncepce testování.

Původní TASysTest běží v prostředí MS Windows na platformě .NET a je naprogramován v jazyce C# v prostředí MS Visual Studio 2013. S testovaným HW komunikuje přes NI VeriStand. TASysTest pro komunikaci s NI Veristandem využívá dostupné .NET API.

Rozšíření je provedeno portací jádra do prostředí NI Real-Time Phar Lap ETS. Samotné jádro je portováno do jazyka C/C++ ve formě 32-bitové Windows DLL, jako nespravovaný kód. Pro vývoj a překlad je využito prostředí MS Visual Studio 2010. Pro spuštění v NI Real-Time Phar Lap ETS je využit nástroj NI VeriStand. Tento nástroj integruje DLL pomocí rozšíření Custom Device.

V závěru práce je provedeno ověření funkčnosti v systému NI Real-Time Phar Lap ETS a v zařízení NI PXIe-8135. Ověření je prováděno simulací chování reálného systému pomocí modelu vytvořeného v NI LabView.

Přínosem práce je odstranění komunikačních latencí mezi TASysTestem a NI VeriStandem. Dalším přínosem je ověření funkčnosti DLL knihovny implementované v jazyce C++ a open source XML parseru v prostředí NI Real-Time Phar Lap ETS. Real-time chování provedené implementace nebylo ověřeno.

Klíčová slova: UPPAAL, TASysTest, Reálný čas, Custom Device, NI Veristand, Integrační testování, National Instruments, Phar Lap ETS, NI PXIe-8135

The aim of this work is and extension of TASysTest Tool about the possibility of running its computing core in real-time environment, while respecting its original testing concept.

Original TASysTest runs on MS Windows platform .NET and is programmend in C# in MS Visual Studio 2013. TASysTest for communication with testing hardware using NI Veristand and for communication with NI VeriStand uses .NET API.

The extension is implemented through porting the kernel to platform NI Real-Time Phar Lap ETS. TASysTest core is portind to C/C++ language in the form 32-bit MS Windows DLL as unmanaged code. For the development and translation is used MS Visual Studio 2010. To launch the DLL in NI Real-Time Phar Lap ETS is used NI VeriStand tool. This tool integrates the DLL by using Custom Device extension of NI VeriStand tool.

At the end of work is performed a verification of system functionality in NI Real-Time Phar Lap ETS in NI PXIe-8135. Verifiaction is performed by simulating the behavior of the real system using a model created in NI LabView.

The benefit of this work is to remove communication latency between TASysTest and NI VeriStand. Another benefit is verifying the functionality of DLL library implemented in C++ and open source XML parser in NI Real-Time Phar Lap ETS.

Real-time behavior performed implementation has not been verified.

Keywords: UPPAAL, TASysTest, Real-Time, Custom Device, NI Veristand, Integration testing, National Instruments, Phar Lap ETS, NI PXIe-8135

Title translation: Real-Time Extension of TaSysTest Tool

Obsah /

Obrázky

Kapitola 1
Úvod

V této kapitole je nejprve popsána motivace automatizování testování elektroniky vozidel, poté je uvedena koncepce původního testovacího prostředí TASysTest a nakonec je uveden cíl práce.

1.1 Motivace

V automobilovém průmyslu je čím dál tím víc kladen důraz na elektronické systémy. Počet a komplexnost těchto systémů ve vozidlech stále narůstá - roste množství komfortních a asistenčních systémů ve vozidlech a elektronikou jsou postupně nahrazovány i základní mechanické systémy. S množstvím a komplexitou jednotlivých systémů rostou časové a tím pádem i finanční nároky na jejich testování. Je prováděno funkční a integrační testování. Při funkčních testech jsou ověřovány všechny individuální funkce komponenty pro každou komponentu zvlášť. Integrační testování spočívá v ověřování funkcí jednotlivých komponent jako celku - testuje se komunikace mezi nimi. Jednotlivé subsystémy a komponenty jsou většinou vyvíjeny současně. Integrační testování je proto prováděno technikou HIL. To umožňuje nalezení chyb již v raném stádiu vývoje, ve kterém většina chyb vzniká. Tím vším nabývá na významu automatizované testování jednotlivých subsystémů ve vozidle. Tento přístup šetří časové a finanční prostředky testování a zvyšuje kvalitu provedených testů odstraněním chyb, které mohou vznikat při manuálně prováděných testech. Čím dříve je chyba objevena, tím snazší je její odstranění. Vývoj v oblasti automatizace generování a provádění testů elektronických systémů vozidel má proto velký význam.

1.2 Testovací prostředí TASysTest

Výchozím bodem je diplomová práce [1]. Výsledkem bylo nové testovací prostředí TASysTest, umožňující generování a provádění model-based integračních testů. Nástroj TASysTest je založen na platformě .NET a je implementován v jazyce C# formou spravovaného kódu.

Schéma původního testovací prostředí je na obrázku 1.1. Koncepce nástroje TASysTest je následující. TASysTest pro svoji práci potřebuje dva další nástroje - nástroj UPPAAL a nástroj NI VeriStand se systémem NI PXIe. Pomocí nástroje UPPAAL je vytvořen model testovaného systému, dále jen model. Model je reprezentován pomocí tzv. časovaných automatů a je exportován ve formátu XML. Tento model tvoří vstupní data pro TASysTest. Nástroj NI VeriStand umožňuje real-time HIL (Hardware-in-the-loop) simulace a spolu se systémem NI PXIe tvoří interface mezi TASysTestem a testovaným HW, dále jen HW. NI VeriStand pro možnost svého vzdáleného ovládání a pro přístup ke svým datům poskytuje .NET API. Toto API je použito pro vzájemnou komunikaci mezi NI VeriStandem a TASysTestem.

Po spuštení TASysTestu je nutné načíst model systému. Paralelně je nutné spustit NI VeriStand projekt a tento spustit v NI PXIe. Po spuštění NI VeriStand projektu je

možné v TASysTestu načíst tzv. Veristand adaptor[1]). Načtení Veristand Adaptoru je nutné. Teprve po úspěšném načtení[2]) Veristand adaptoru je možné zahájit testování. Před inicializací je provedeno předzpracování vybraných částí načteného modelu pro rychlejší vykonávání testu.

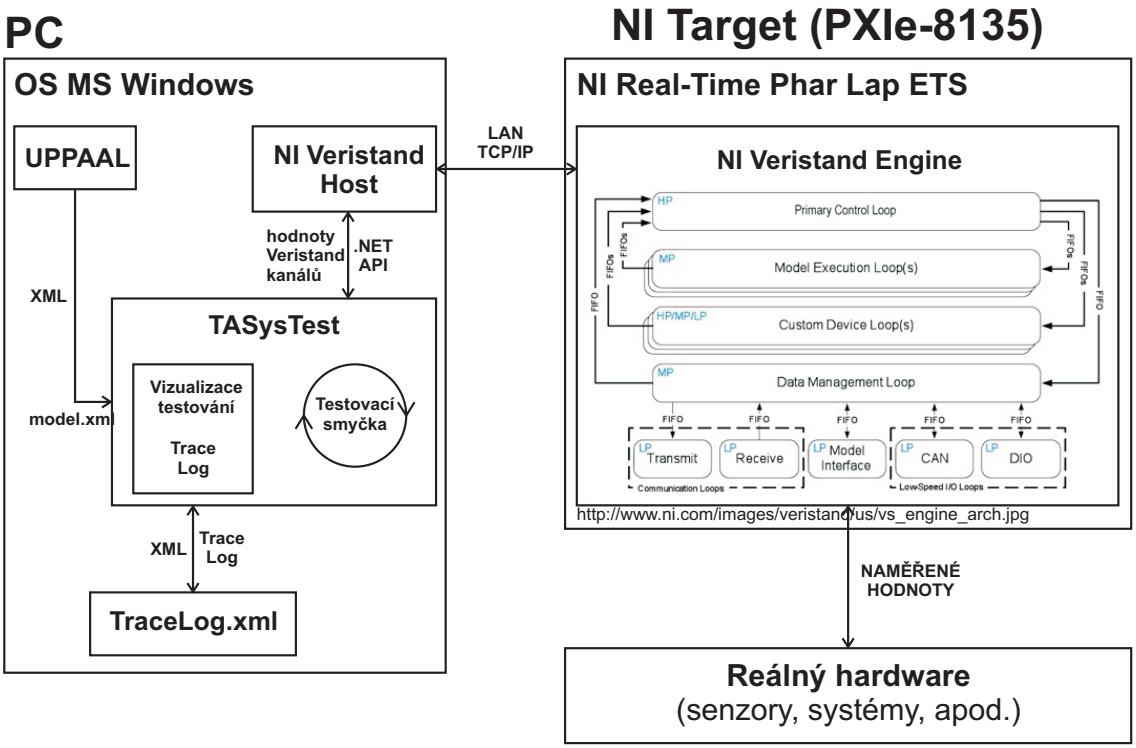
Testování je možné provádět buď po jednotlivých krocích manuálně nebo automaticky spuštěním testu. Po zahájení testování TASysTest vždy zhodnotí možné kroky a z nich jeden vybere podle zvolené strategie[3]) testování, ten vykoná posunutím modelu do nového stavu a zkontroluje validitu invariantu nového stavu modelu. Pokud není invariant splněn nebo pokud není dostupný žádný[4]) krok testu, testování je ukončeno. Automatické testování probíhá ve smyčce o určité frekvenci. Tuto Hodnotu frekvence je možné nastavit před spuštěním testu. Všechny kroky testu jsou ukládány do řádků logu. Z těchto záznamu je možné zpětně prohlížet průběh testu.

Nástroj UPPAAL, potažmo TASysTest předpokládá pro korektní běh testovacího algoritmu ekvidistantní[5]) časové intervaly jednotlivých iterací testovací smyčky. TASysTest běží pod OS MS Windows od nějž je odvozeno i časování testovací smyčky. MS Windows není operační systém, který provádí operace v reálném čase, tzn. negarantuje odezvu aplikace v předem definovaném časovém intervalu. Kromě nepřesnosti časování použitého operačního systému jsou přítomny i další časové nepřesnosti způsobené např. latencí komunikace mezi TASysTestem a HW prostřednictvím NI VeriStandu, latencí I/O portů, nestabilitou frekvence hodin, atd. Od výsledku testování je nutné tyto nakumulované hodnoty odečíst.

Testovací proces je možné přiblížit k deterministickému časování provedením některých předběžných opatření[6]) - vypnutí všech nekritických služeb na pozadí, nastavení vysoké priority pro proces, ve kterém TASysTest běží, ukončit všechny nepotřebné aplikace a vypnout všechny úspory energie. Avšak i při dodržení uvedených opatření, současné testovací prostředí neposkytuje zcela korektní časování, pracuje pouze v tzv. best-effort režimu[7]). Tím vznikají nepřesnosti a je snížena maximální testovací frekvence.

Bližší popis testovacího prostředí TASysTest je uveden v [1].

[1]) Poskytuje spojení vybraných typů proměnných z domény typů proměnných nástroje UPPAAL mezi TASysTestem a NI VeriStandem využitím .NET API poskytovaného NI Veristandem.
[2]) Po úspěšném navázání spojení a spárování kanálů.
[3]) Původní verze TASysTestu implementovala pouze náhodnou strategii, postupem času byly přidány systematická a experimentální strategie.
[4]) dojde-li k uváznutí vzhledem k návrhu modelu
[5]) zachovávající konstantní vzdálenost
[6]) Čerpáno z [1], odstavec 3.2.4, str.24
[7]) Jinými slovy, má největší snahu časování dodržet, ale přesné časování není zaručeno.



Obrázek 1.1. Schéma původního testovacího prostředí TASysTest.

1.3 Cíl práce

Cílem práce je přenesení výpočetního jádra TASysTestu do prostředí, které je schopno zajistit real-time časování. Hlavním přínosem by mělo být dosažení deterministického časování pro jeden krok testu. Dalším přínosem by mohlo být zvýšení testovací frekvence a odstranění nutnosti provádět korekce latencí vzniklých komunikačním řetězcem TASysTest - Veristand .NET API - NI Veristand - HW.

Výsledkem práce bude rozšíření stávajícího nástroje TASysTest o real-time výpočetní jádro, které poběží přímo v NI PXIe targetu[1]) pod operačním systémem NI Real-Time Phar Lap ETS. Toto rozšíření bude používat pro svoji práci nástroje UPPAAL a NI VeriStand. Nově nebude TASysTest využívat NI VeriStand pro komunikaci s testovaným HW prostřednictvím poskytovaného .NET API, ale výpočetní jádro TASysTestu bude vhodným způsobem rozšiřovat funkce NI Veristandu, tzn. že výpočetní jádro TASysTestu poběží přímo ve výkonné smyčce NI Veristandu - běh testu a komunikace s HW budou sloučeny do jednoho systému, čímž dojde k odstranění latencí komunikace mezi TASysTestem a NI Veristandem. Výsledky testu budou po skončení testu dostupné na cílové platformě ve formě logu. Log bude mít formát kompatibilní s formátem logu TASysTestu. Současný nástroj TASysTest bude sloužit pro vizualizaci výsledku testu. Vizualizaci bude možné provést manuálně po skončení testu ze zaznamenaného logu a to na platformě MS Windows.

Na začátku práce bude proveden návrh řešení a výběr vhodných nástrojů pro jeho realizaci. Následně bude provedena implementace řešení. Závěrem práce bude provedeno ověření funkčnosti a provedené implementace na operačním systému NI Real-Time Phar Lap ETS a v zařízení NI PXIe-8135. Nakonec bude uveden návrh možných rozšíření.

[1]) PXIe je koncové konkrétní zařízení, v našem případě NI PXIe-8135.

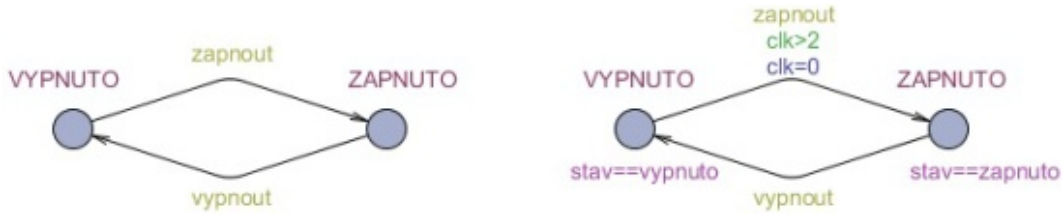
Kapitola 2
Použité nástroje a technologie

Tato kapitola obsahuje popis zvolených nástrojů a technologií použitých pro portaci jádra TASysTestu do prostředí real-time a pro běh výsledného nástroje.

2.1 Časované automaty

2.1.1 Úvod

Koncept časovaných automatů je použit pro popis modelu testovaného systému, který tvoří vstupní data pro testovací nástroj. Časované automaty vychází z tzv. deterministických stavových automatů (finite state automata), které jsou rozšířeny přidáním popisu časového toku a s tím související možnosti kontrolování časování.



Obrázek 2.1. Ukázka rozdílu mezi deterministickým stavovým automatem(vlevo) a časovaným stavovým automatem(vpravo).

Zjednodušeně, konečný stavový automat je vyjádřet formou orientovaného grafu s množinou uzlů/stavů a množinou orientovaných hran mezi nimi, které umožňují přechody mezi stavy. V tomto případě je možné hrany považovat za příkazy k přechodu.

Ukázka jednoduchého konečného stavového automatu je na obrázku 2.1 vlevo. Automat na obrázku reprezentuje jednoduchý vypínač. Vypínač má dva stavy - VYPNUTO a ZAPNUTO. Mezi stavy je možné přecházet pomocí příkazů zapnout a vypnout. Kromě orientace hran není v popsaném systému žádné omezení. Tzn. všechny hrany, které vedou z konkrétního uzlu je možné projít.

Na obrázku 2.1 vpravo je původní konečný stavový automat rozšířen na časovaný automat. To je provedeno přidáním času do systému, v tomto případě proměnné clk typu čas. S každým pohybem v grafu se automaticky zvýší hodnota časových proměnných. Na základě toho je možné definovat omezení, které zmenší množinu hran, po kterých je možné z daného uzlu přejít. V tomto případě musí být čas větší než 2, aby bylo možné po hraně zapnout přejít ze stavu VYPNUTO do stavu ZAPNUTO. Při přechodu je možné vykonat ještě operaci přiřazení, v tomto případě se jedná o nastavení proměnné clk do nuly.

Časovaný automat rozšiřuje popis konečných stavových automatů nejen o tok času, ale i o další možnosti, jako je například kontrola invariantu stavu, do kterého se přechází. Pokud má stav, do kterého se přechází definovaný invariant, musí být po přechodu do tohoto stavu splněn, jinak je procházení ukončeno, resp. do takové stavu není možné

přejít. Na obrázku 2.1 vpravo je definovaný invariant v obou stavech. Ve stavu VYPNUTO musí být hodnota proměnné stav rovna hodnotě vypnuto (např. logická 0) a ve stavu ZAPNUTO musí být hodnota stavu rovna hodnotě zapnuto (např. logická 1).

2.1.2 Formální popis

Čerpáno z [1], kapitola 2.1.

2.1.2.1 Konečný stavový automat

Konečný stavový automat A je pětice:

$$A = (Q, \Sigma, \delta, q_0, F)$$

kde:

- Q je konečná neprázdná množina stavů,
- Σ je konečná neprázdná množina vstupních symbolů, nazývaná abeceda,
- δ je tzv. přechodová funkce $\delta : Q \times A \to Q$, popisující pravidla přechodů mezi stavy. Jedná se o množinu QxΣ, která popisuje přechod p jako $p = \delta(q, a) : p, q \in Q, a \in \Sigma$, což znamená: když je automat ve stavu q a symbol a je čten ze vstupu, nový stav bude p,
- q_0 je počáteční stav, $q_0 \in Q$,
- F je množina finálních stavů, $F \subset Q$, dosáhne-li automat jednot ze stavů z množiny F, akceptoval vstupní řetězec.

2.1.2.2 Časovaný automat

Časované automaty upravují základní koncept konečných stavových automatů přidáním hodin pro reprezentaci plynutí času a dále ho rozšiřují zavedením několika výrazů pro přechody, formálně: časovaný automat je definován jako množina akcí A a množina hodin C jako čtveřice:

$$(A, C) = (L, l_0, lE)$$

kde:

- L je konečná množina lokací (ekvivalent k množině stavů Q konečného stavového automatu),
- l_0 je počáteční lokace (ekvivalent q_0 v konečném stavovém automatu),
- l je množina invariantů, které jsou přiřazeny lokacím L,
- E je množina hran(přechodů) mezi lokacemi.

kde hran je pětice:

$$E = (l, q, a, r, l^t)$$

kde

- l je zdrojová lokace hrany,
- g je tzv. quard, výraz, který musí být splněn, aby mohla být hrana aktivní,
- q je tzv. akce, kde postifový znak je buď '!', který označují výstupní(vyvolávající) akci, nebo '?', který označuje vstupní(přijímací) akci,
- r je množina hodin, které mají být resetovány do 0,
- l^t je cílová lokace hrany.

2.1.2.3 Užití automatů pro modelování systémů

Modelování systémů pomocí časovaných automatů se provádí tak, že se definují všechny možné stavy, ve kerých se může reálný systém nacházet. Poté se přidají přechody mezi těmito stavy. Přechody je možné omezit pomocí přechodových podmínek a výrazů a pomocí invariantů v jednotlilvých stavech.

Controller-observer

Dvojice automatů. Jeden controller, jehož funkcí je poskytování vstupů systému a jeden nebo více observer automatů. Observer automat monitoruje stavy systému a kontroluje přechodové podmínky a invarianty. U reálných systémů může observer automat provádět porovnání virtuálního a skutečného stavu testovaného systému.

2.2 UPPAAL

2.2.1 Popis

Zkratka z Uppsala University in Sweden a z Aalborg University in Denmark, jejichž spoluprácí je tento nástroj vyvíjen. Jedná se o integrované vývojové prostředí pro modelování, validaci a verifikaci systémů reálného času reprezentovaných časovanými automaty, které jsou rozšířeny o datové typy.[1]).

Použitá verze nástroje je UPPAAL 4.0.14. Ve výsledku práce je tento nástroj využit pro vytváření systému modelů testovaných systémů[2]) jenž tvoří vstupní data pro testovací nástroj.

2.2.2 Formát popisu systému - struktura XML dokumentu

Jeden systém se skládá z jednoho až z několika modelů zvaných ‘Template’.

UPPAAL podporuje tři formáty reprezentace modelů mezi nimiž je i formát XML, který nabízí nejvíce možností popisu systému a proto je primárně využíván. Zároveň zaručuje dobrou přenositelnost mezi platformami. Ostatní dostupné formáty jsou již zastaralé - nejsou dále podporovány a neposkytují multiplatformnost. Hierarchie XML dokumentu, kterou je nutné uvažovat při načítání popisu systému do real-time verze výkonného jádra TASysTestu je zobrazena na obrázku 2.2.

[1]) volně přeloženo a upraveno z [7]
[2]) stejně jako v původní verzi TASysTestu

```
<nta>
        <declaration></declaration>
        <template>
                <name></name>
                <declaration></declaration>
                <parameter></parameter>
                <location id="" x="" y="">
                        <name></name>
                        <label kind="invariant"></label>
                        <label kind="comments"></label>
                </location>
                <init ref="location_id"/>
                <transition>
                        <source ref="location_id"/>
                        <target ref="location_id"/>
                        <label kind="synchronisation"></label>
                        <label kind="guard"></label>
                        <label kind="assignment"></label>
                        <nail x="" y=""></nail>
                </transition>
        </template>
        <system></system>
</nta>
```

Obrázek 2.2. Struktura XML dokumentu, která je načítána do TASysTestu. Zvýrazněné tagy se vyskytují opakovaně. Tagy s názvem label se vykytovat nemusí vůbec.

Formálně se popis UPPAAL systému skládá z těchto částí:[1])

- **Global declarations** (v XML: nta/declaration)

  Globální deklarace. Obsahuje všechny globální deklarace. Pokud není v nějakém templatu jmenný ekvivalent proměnné, jsou tyto proměnné sdíleny mezi všemi templaty.

- **System declarations** (v XML: nta/system)

  Systémová deklarace. Slouží pro specifikaci systému jako celku. Zde se jednotlivé templaty propojují pomocí jejich parametrů a globální proměnných. Všechny templaty, které mají být testovány musí být zde instancovány.

- Několik **template** (v XML: nta/template)

  - Každý template se skládá z následujících položek:

    - **name** (v XML: nta/template/name)

      Jméno templatu. Využito při instancování templatuů

    - **local declaration** (v XML: nta/template/declaration)

      Lokální deklarace. Specifikuje lokální instanční proměnné.

[1]) volně přeloženo a upraveno z [1]

- **location** (v XML: nta/template/location)

  Lokace. Vícenásobný výskyt v xml dokumentu. Lokace je stav/uzel modelu reprezentovaného tímto templatem. Každý stav má unikátní id, kterým je označen. Volitelně může mít uživatelem definovaný popisek(name). Dále může mít definovaný invariant(invariant) a komentář(comments). Invariant je podmínka, která musí být splněna, pokud je systém ve stavu s definovaným invariantem. Komentář slouží pro předání dalších údajů do systému, např. parametr relevance, který slouží pro vyhledávací strategii.

- **initial location** (v XML: nta/template/init)

  Počáteční lokace. Defaultní stav. V parametru obsahuje referenci(ref) na počáteční stav systému.

- **transitions** (v XML: nta/template/transition)

  Přechod, orientovaná hrana. Vícenásobný výskyt v xml dokumentu. Obsahuje zdrojový(source ref="") a cílový(target ref="") uzel, jehož id je definováno parametrem ref. Dále volitelně obsahuje položky synchronisation, guard a assignment. Tyto položky jsou definovány tagem label jeho parametrem kind. Tyto volitelné položky slouží jako kontrolní mechanizmus přechodu po hraně. Jejich význam je následující

  - **Synchronisation channel** ('Sync')

    Synchronizační kanál. Podle postfixového znaku: buď výstupní ('!') synchronizační hrana nebo vstupní ('?') synchronizační hrana. Jedná se o komplementární hrany. Aby bylo možné projít přes synchronizační hranu, je nutné, aby pro každou výstupní hranu existovala alespoň je vstupní hrana a opačně, aby pro každou vstupní hranu existovala alespoň jedna výstupní hrana. Tento mechanizmus slouží pro synchronizaci jednotlivých modelů v systému.

  - **Assignment** ('Update')

    Obsahuje skript, který je při přechodu přes hranu vykonán. Jednotlivé příkazy musejí být oddělovány čárkou (nikoliv středníkem).

  - **Condition** ('Guard')

    Obsahuje podmínku, kterám usí být splněna, aby mohlo být přejito přes hranu. Pokud podmínka splněna není, hrana je neaktivní a nemůže být přes ni přejito.

  - **Selection** ('Select')

    Slouží pro svázání proměnné s nějakým rozsahem.

  Uvedené vlastnosti využívají built-in skript nástroje UPPAAL.

2.2.3 Příklad popisu systému

V této sekci je uveden příklad časovaného automatu, který je použit pro testování základní funkčnosti nově implementovaného jádra TASysTestu. Jedná se o UPPAAL system pro měření odezvy systému. Konkrétně se jedná o jednoduchý systém pro rozsvícení a zhasnutí světla pomocí vypínače. Skládá se z dvojice modelů controller-observer. Controller je vypínač a observer je světlo.

2.2.3.1 Model vypínače

Na obrázku 2.3 je znázorněn jednoduchý model, který reprezentuje obyčejný vypínač. V popisu systému reprezentuje roli controllera. Skládá se ze tří stavů.

- **nepojmenovaný, počáteční stav**

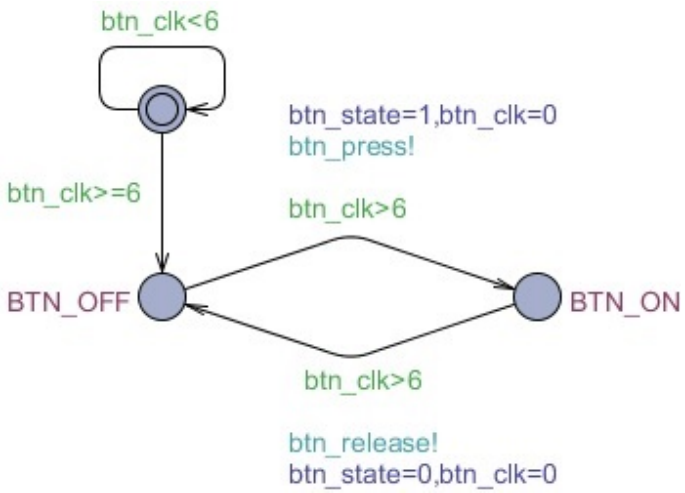
  První stav, nepojmenovaný. V tomto modelu se jedná o tzv. počáteční stav a v modelu je označen dvěma soustřednými kruhy. Reprezentuje počáteční stav systému. Vzhledem k omezujícím časovým podmínkám na hranách by v případě absence tohoto stavu došlo v tomto systému k uváznutí, k deadlocku (nebyla by aktivní žádná hrana, kterou by bylo možné projít).

- **BTN_OFF**

  Tento stav představuje vypnutý stav vypínače.

- **BTN_ON**

  Tento stav reprezentuje zapnutý stav vypínače.



Obrázek 2.3. UPPAAL Model vypínače - role Controller

2.2.3.2 Model světla

- **START_NODE**

  Počáteční stav modelu.

- **SWITCHING_ON**

  Stav zapínání světla,přepínání ze stavu LIGHT_OFF do stavu LIGHT_ON. Díky němu je možné simulovat určité zpoždění sepnutí světla.

- **LIGHT_ON**

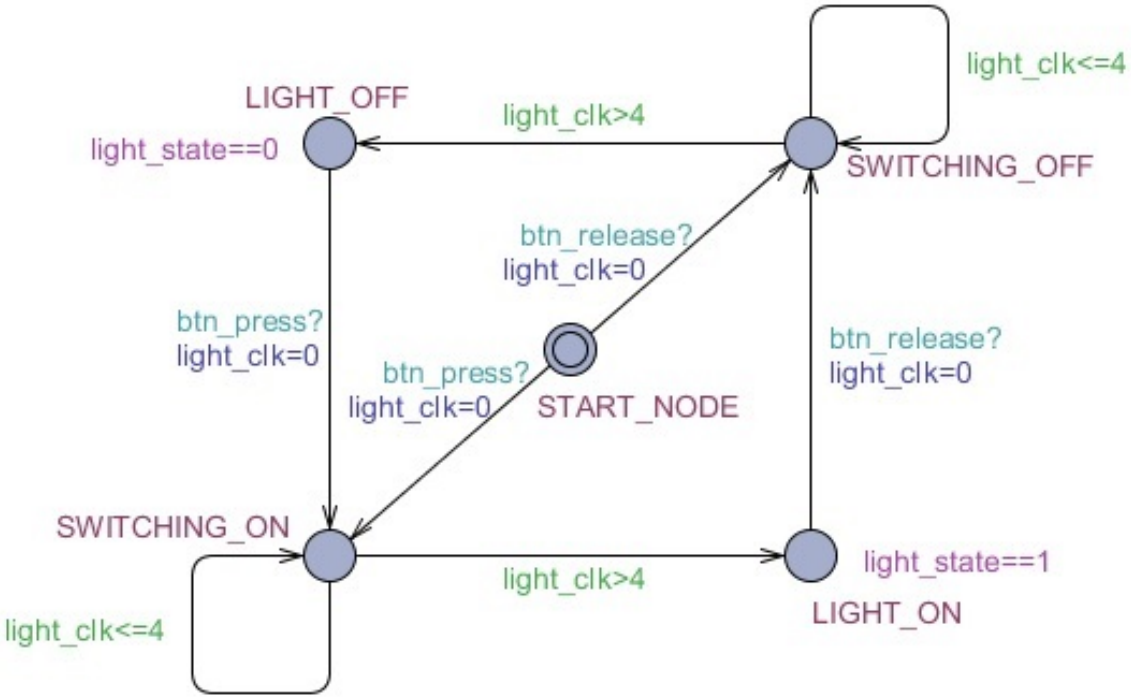
  Stav zapnutého světla.

- **SWITCHING_OFF**

  Stav vypínání světla, přepínání ze stavu LIGHT_ON do stavu LIGHT_OFF. Díky němu je možné simulovat určité zpoždění vypnutí světla.

- **LIGHT_OFF**

  Stav vypnutého světla.



Obrázek 2.4. UPPAAL Model světla - role Observer

2.2.3.3 Provedení jednoho kroku v systému

Jednotlivé modely musejí být v systému instancovány a propojeny vstupy a výstupy závislých instancí. U každého modelu jsou jeho vstupy a výstupy definovány pomocí referencí. Propojení se provádí pomocí globálních proměnných.

V modelu vypínače, pro přejití z nepojmenovaného stavu do stavu BTN_OFF musí být hodnota hodin btn_clk větší nebo rovno hodnotě 6. Délka doby závisí na rychlosti provádení testu. Hodnota 6 znamená 6 kroků simulace. Pro přejití ze stavu BTN_OFF do stavu BTN_OFF je opět nutné, aby hodnota hodin btn_clk byla vyšší než 6 ale zároveň musí existovat alespoň jedna vstupní (?) synchronizační hrana v jiném instancovaném modelu v systému s názvem synchronizace ‘btn_press?’. Pokud taková hrana existuje a pokud jsou splněny všechny omezení na obou komplementárních hranách, je možné po této hraně přejít. Současně bude přejito i přes komplementární hranu v jiném modelu v systému. Po přejití touto hranou se provedou veškerá přiřazení na ní i na odpovídající komplementární hraně. V případě modelu světla, který je na obrázku 2.4, bude v případě přejití modelu vypínače ze stavu BTN_OFF do stavu BTN_ON přejito z uzlu START_NODE do uzlu SWITCHING_ON.

2.2.3.4 XML reprezentace modelu vypínače

```
<template>
	<name x="8" y="8">btn</name>
	<parameter>chan &amp;amp; btn_press, chan &amp;amp; btn_release, int &amp;amp;
	btn_state</parameter>
	<declaration>// Place local declarations here clock btn_clk;</declaration>
	<location id="id0" x="-608" y="-512"></location>
	<location id="id1" x="-416" y="-416">
		<name x="-400" y="-424">BTN_ON</name>
	</location>
	<location id="id2" x="-608" y="-416">
		<name x="-688" y="-424">BTN_OFF</name>
	</location>
	<init ref="id0"/>
	<transition>
		<source ref="id0"/>
		<target ref="id0"/>
		<label kind="guard" x="-640" y="-568">btn_clk&lt;6</label>
		<nail x="-640" y="-512"/>
		<nail x="-640" y="-544"/>
		<nail x="-576" y="-544"/>
		<nail x="-576" y="-512"/>
	</transition>
	<transition>
		<source ref="id0"/>
		<target ref="id2"/>
		<label kind="guard" x="-688" y="-480">btn_clk&gt;=6</label>
	</transition>
	<transition>
		<source ref="id1"/>
		<target ref="id2"/>
		<label kind="guard" x="-536" y="-384">btn_clk&gt;6</label>
		<label kind="synchronisation" x="-544" y="-352">btn_release!</label>
		<label kind="assignment" x="-544" y="-336">btn_state=0,btn_clk=0</label>
		<nail x="-512" y="-384"/>
	</transition>
	<transition>
		<source ref="id2"/>
		<target ref="id1"/>
		<label kind="guard" x="-544" y="-472">btn_clk&gt;6</label>
		<label kind="synchronisation" x="-544" y="-504">btn_press!</label>
		<label kind="assignment" x="-544" y="-520">btn_state=1,btn_clk=0</label>
		<nail x="-512" y="-448"/>
	</transition>
</template>
```

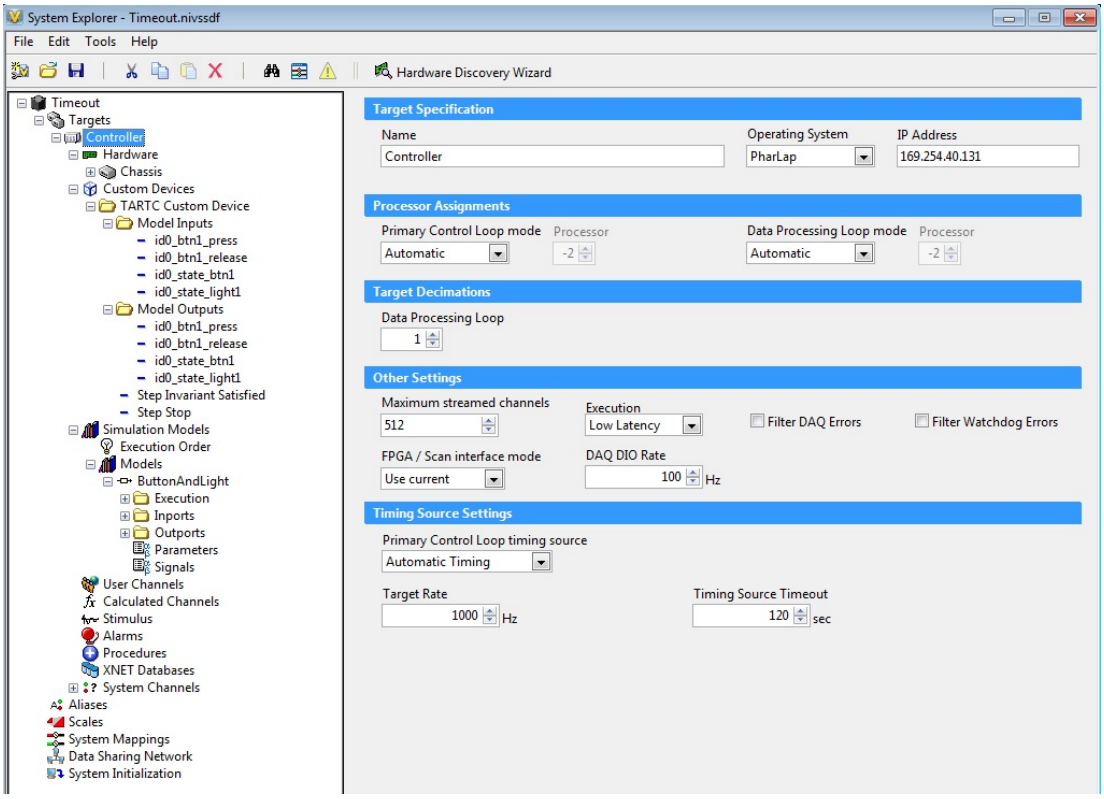
Obrázek 2.5. XML reprezentace modelu vypínače

2.3 NI VeriStand

2.3.1 Co je NI Veristand

Následující odstavec je volně přeložen z [10].

Veristand je softwarové prostředí pro konfiguraci aplikací testování v reálném čase. Umožňuje provádění operací jako generování stimulů, vysokorychlostí získávání dat, poskytuje kalkulované kanály a možnost provádět vlastní škálování. Dále umí importovat řídící algoritmy, simulační modely a jiné aplikace z LabVIEW i z jiných vývojových prostředí. S těmitoo procesyje možné komunikovat pomocí poskytnutého uživatelského rozhraní, které zahrnuje nástroje pro zápis dat, monitorování alarmů, I/O kalibrace a stimulační profily. Pokud nejsou poskytnuté funkce dostatečné, je možné VeriStand přizpůsobit a rozšířit o další funkčnost pomocí jiných softwarových prostředí jako je LabVIEW, ANSI C/C++, MathWorks a další, které jsou určeny pro modelování a programování. Prostředí je zobrazeno na obrázku 2.6.



Obrázek 2.6. Softwarové prostředí NI VeriStand - System explorer, system definition file. Na obrázku je vidět stránka, na které je prováděno nastavení RT targetu. Vlevo je vidět hierarchie celého systému.

2.3.2 Custom Device [5]

Jak je uvedeno výše, VeriStand nabízí mnoho možností rozšíření jeho funkcionality. Například pomocí modelu zkopilovaného v LabVIEW. Jednou z možností jeho rozšíření je tzv. Custom Device. Možnosti rozšíření se od sebe liší způsobem běhu ve VeriStand enginu. Custom Device je stejně jako model vyvíjeno v LabVIEW. Oproti modelu však nabízí větší modularitu. Pro jeho vývoj je poskytnut speciální framework.

2.4 Programovací jazyk a vývojové prostředí

Jako učebnice programování byla využita tato práce [4]. Návod pro sestavení dynamické knihovny [14].

Portace nástroje TASysTest do prostředí real-time bylo prováděno pomocí Custom Device a v něm integrované DLL knihovny. Pro vývoj knihovny bylo nutné zvolit vhodný programovací jazyk. Původní TASyTest byl implementovaný v jazyce C#. C# kód je tzv. spravovaný. Pro svůj běh využívá Microsoft .NET Framework. Takový kód není kompilovaný přímo do strojového kódu, ale je kompilovaný do tzv. Intermediate Language (IL) a poté je vykonávaný pomocí interpretru. Spravovaný kód, anglicky Managed Code, zajišťuje správu paměti a typovou kontrolu. Takový kód však není v NI Real-Time Phar Lap ETS podporovaný. Proto bylo nutné zvolit jiný programovací jazyk - nabízel se jazyk C++ a jazyk ANSI C. Vzhledem k tomu, že C# je objektově orientovaný jazyk, byl jasnou volbou programovací jazyk C++, jeho nespravovaná verze.

Pro vývoj bylo dále nutné zvolit vývojové prostředí. National Instruments podporuje běh DLL knihoven zkopilovaných ve dvou prostředích, které podporují programovací jazyk ANSI C nebo C++. Pro vývoj v jazyce ANSI C nabízí nástroj NI LabWindows CVI. Druhým prostředím, jímž zkompilované knihovny jsou firmou NI v real-time zařízeních podporovány je prostředí Microsoft Visual Studio 2010, runtime support 1.1. Tato verze je v současnoti nejvyšší, která je pro vývoj knihoven podporována. Jelikož NI LabWindows podporuje pouze jazyk ANSI C a nepodporuje většinu dostupných knihoven[1]), bylo zvoleno prostředí Microsoft Visual Studio.

[1]) například žádný open source XML parser

Kapitola 3
Návrh řešení

V této kapitole jsou shrnuty požadavky na portované jádro TASysTestu a dále je zde popsaná nová architektura výsledného výpočetního systému.

3.1 Požadavky na výsledný nástroj

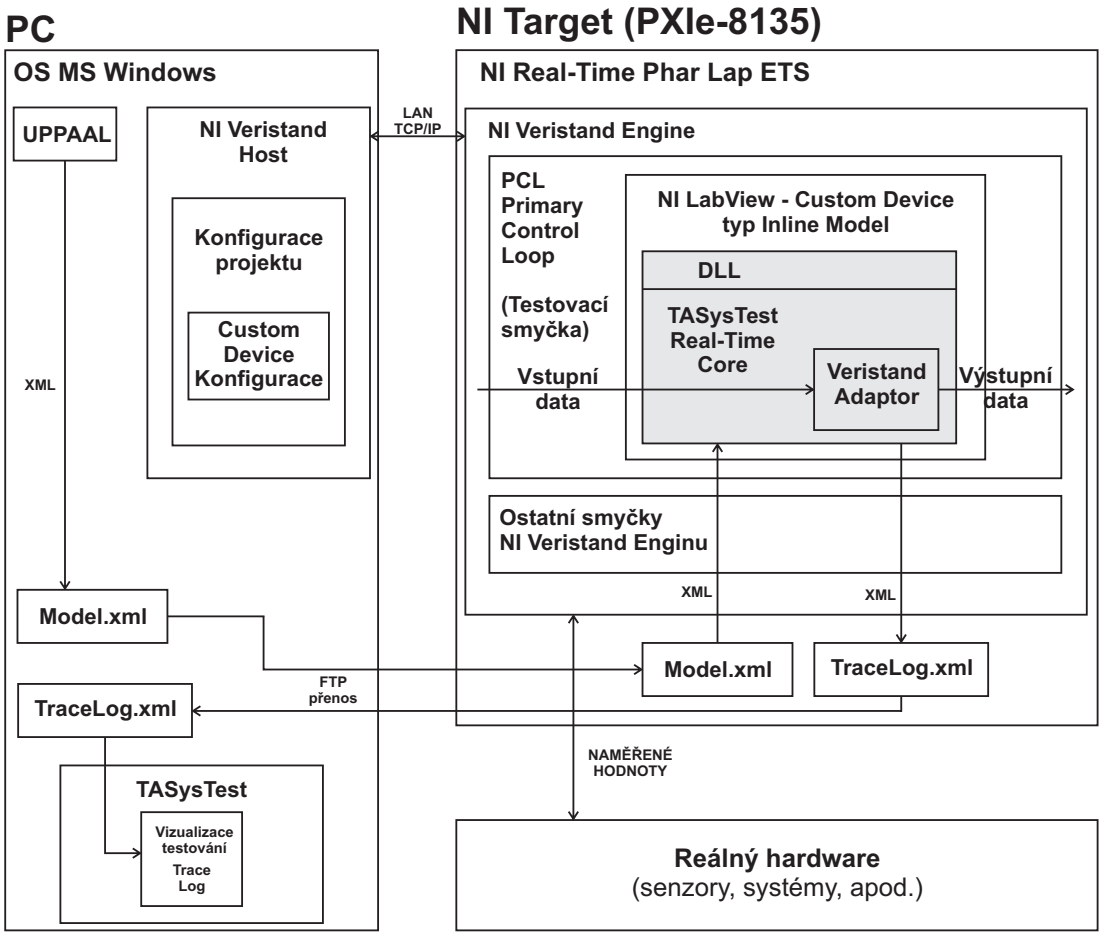
Prostředí reálné času je NI Real-Time Phar Lap ETS. Ze zadání musí být zajištěna komunikace s NI VeriStandem. Hlavním požadavkem na výsledný nástroj je dosažení optimálního časování s minimálními odchylkami[1]). Současným požadavkem je provedení rozšíření stávajícího nástroje TASysTest s minimálními[2]) změnami jeho použití. Ideální by bylo provést změny takové, aby bylo možné před startem testu v TASysTestu možné vybrat platformu, která bude vykonávat hlavní testovací smyčku - MS Windows nebo NI Real-Time Phar Lap ETS.

Dalším požadavkem je provedení portace současné implementace do prostředí LabView Real-Time tak, aby byla zachována stejná struktura a modularita zdrojového kódu, která by odpovídala[3]) původní implementaci - používáním implementace TASysTestu je vyzkoušeno, že původní implementace je přehledná a snadno rozšiřitelná o další funkčnost.

3.2 Úpravy architektury

Na obrázku 3.1 je schéma navržené architektury. Úpravy architektury jsou provedeny takovým způsobem, aby byl zajištěn běh přeportovaného jádra v prostředí real-time a tak, aby bylo možné komunikovat s NI Veristandem.

[1]) např. odchylky způsobené komunikací pomocí .NET API, latencí I/O operací, apod.
[2]) nejlépe žádnými :)
[3]) samozřejmě v rámci možností a odchylek původních a nově použitých implementačních nástrojů a běhových prostředí



Obrázek 3.1. Nově navržená architektura, která vznikla z původní architektury 1.1

Nová architektura funguje následovně. Základem je DLL knihovna, ve které je přeportované výkonné jádro TASysTestu. Toto jádro je integrováno pomocí frameworku Custom Device, který zprostředkovává schopnost běhu knihovny ve VeriStandu. Jádro TASysTestu se pomocí Custom Device stává součástí NI Veristandu a tím rozšiřuje jeho funkce. Uvnitř Custom Device je prováděno dynamické generování kanálů podle načteného testovaného modelu. Tyto kanály slouží pro komunikaci s ostatními částmi NI VeriStandu. Jádro TASysTestu je tímto způsobem možné propojit s libovolnou částí VeriStandu a je možné využít libovolných jeho nástrojů pro jejich stimulaci. Formát vstupních a výstupních dat TASysTestu je zachován.[1]) To umožňuje generovat testované modely nástrojem UPPAAL. Dále je tím umožněno využít TASysTest pro prohlížení a zpětné krokování provedeného testu.

Zásadní změnou oproti původní architektuře je způsob komunikace s NI Veristandem. V původní architektuře bylo s tímto nástrojem komunikováno pomocí .NET API, zatímco v nové architektuře je výpočetní jádro TASysTestu přímo součástí VeriStandu. Tím nová architektura odstraňuje komunikační latence mezi TASysTestem a NI VeriStandem.

[1]) viz. 4.1

Kapitola 4
Implementace

Tato kapitola popisuje implementaci portace výpočetního jádra TASysTestu do prostředí real-time.

4.1 Výpočetní jádro TASysTestu

4.1.1 Načtení modelu

V prní fázi implementace bylo nutné provést načtení vstupních dat. Vstupní model systému je reprezentovaný ve formátu XML. Pro jeho načtení bylo nutné použít XML parser. Bylo vyzkoušeno mnoho open source xml parserů jako Xerces, libxml2, expat, atd. Nakonec byl zvolen TinyXML-2 [6] parser. Hlavním důvodem pro jeho volbu byl fakt, že zkompilovaný do DLL knihovny pomocí MS Visual Studia 2010 fungoval v prostředí reálného času. Navíc se snadno používá a snadno začleňuje do projektu - nemá žádné závislosti na jiné knihovny. Jedná se o DOM XML parser. Podporuje UTF-8 kódování. Nepodporuje žádné pokročilé XML funkce, avšak pro naše účely to nevadilo. Hledisko paměťové a časové náročnosti není důležité, je používán ve fázi načítání modelu a na běh testu nemá žádný vliv. Je disribuován pod zlib licencí[1]).

4.1.2 Výkonné jádro TASysTestu

V další fázi byla prováděna implementace výkonného jádra původního TASysTestu. V úvodu práce byl kladen požadavek na minimální odchylky od původní implementace TASysTestu. Jednak proto, že je původní implementace přehledná a snadno rozšiřitelná a pak kvůli další možné rozšiřitelnosti kódu. Pokud by měla být provedena nějaká změna základní funkčnosti výpočetního jádra je jednoduší ji provést na dvou místech stejným způsobem. Proto byla implementace prováděna formou přepisu výkonného jádra TASysTestu ze spravovaného jazyka C# do nespravované verze jazyka C++. Odchylky v implementaci vznikly odlišnostmi těchto jazyků.

4.1.3 Veristand Adaptor

Pro výmenu dat mezi NI VeriStandem a DLL knihovnou byl implementován Veristand Adaptor. Jednoduchá třída, která tvoří rozhraní mezi jádrem TASysTestu a kanály VeriStandu. Vytváří bufferování jednoho stavu všech proměnných. NI Veristand prostřednictvím Custom Device čte a zapisuje data z a do VeristandAdaptoru. Jádro TASysTestu si je poté z adaptoru vyčítá nebo je do něj zapisuje. Aby nedošlo k souběhu přístupu, pro přístup ke společným datům jsou poskytnuty zámky ve formě proměnných. Z principu integrace knihovny v Custom Device by však nemělo k souběhu vůbec dojít.

[1]) znění zlib licence http://tringi.trimcore.cz/zlib_Licence

4.1.4 Globální správa paměti

Implementace je prováděna v jazyce C++, ve kterém se musí programátor starat o správné přidělování a následné uvolňování paměti. Modelování systému je prováděno pomocí tříd. Třída je zobecnění běžný datový typ. Definuje členské proměnné a operace nad nimi. Pro konstrukci a destrukci využívá několik metod. Disponuje konstruktorem a destruktorem. Tyto metody jsou používány při klasické manuální konstrukci nových instancí. Dále také disponuje kopírovacím konstruktorem a operátorem přiřazení. Tyto jsou použity při vytváření instance z jiné již existující. Pokud vzniká nová instance, je volán kopírující konstruktor. Pokud nevzniká nové místro v paměti, je volán operátor přiřazení. Při vzniku objektu získá třída zodpovědnost za svoje členské proměnné. Pokud se jedná o staticky alokované proměnné, programátor se nemusí o jejich destrukci starat. Pokud se jedná o dynamicky alokované proměnné, musí programátor explicitně zajistit jejich správnou konstrukci a destrukci. Pokud existuje dynamicky alokovaná členská proměnná, je nutné explicitně definovat výše uvedené speciální metody třídy. Resp třída by měla být navržena podle pravidla nuly nebo tří. Buď aby nebylo nutné explicitně definovat kopírující konstruktor, operátor přiřazení a destruktor nebo je nutné implementova je všechny. Z toho plyne, že při správném návrhu třídy, přebírá třída zodpovědnost za destrukci svých členských objektů. Tím by mělo být zajištěno, že nebude docházet k memory leakům.

Pokud se dodrží výše uvedené principy, pokaždé, když provedeme přiřazení objektů mezi třídami, musíme nějakým způsobem rozhodnout, která třída za přiřazený objekt přebere zodpovědnost. Tím vzniká zmatek, kde je vlastně objekt uložen.

Testovaný systém je reprezentován modely a každý model je popsán řekněme orientovaným grafem. Každý model je reprezentován templatem a ten má označen výchozí a právě aktivní uzel. Testování probíhá tak, že se prochází jednotlivé grafy a odkazy na právě aktivní uzel se mění. Pokud by se při každé změně přehazovala zodpovědnost z instance na instanci vznikl by zmatek. Bylo by možné pokaždé provádět destrukci a konstrukci objektu, ale to by bylo neefektivní.

Řešením je vytvořit vlastní správu paměti. K tomu účelu byla navržena třída statická třída TAMemory. Třída TAMemory je implementována tak, že pro každý druh proměnné je vytvořen vector pointrů. Koncept je takový, že při vzniku instance se parametrem rozhodne, zda je daný objekt spravovaný nebo ne. Pokud není spravovaný, přebírá za něj zodpovědnost objekt, který ho vytvořil. Pokud však spravovaný je, přebírá za něj zodpovědnost třída TAMemory. Přiřazení odkazu na instanci spravované proměnné probíhá v konstruktoru proměnné pomocí registračních metod, které třída TAMemory poskytuje. Proměnné se přidávají do třídy TAMemory průběžně a jejich uvolnění je provedeno na konci programu. Tato koncepce umožňuje snadné předáváné objektů pointrem. Nevzniká zmatek o tom, která třída má vlastně provést destrukci objektu. Přístup je navržen pro proměnné, které vznikou na začátku spuštění knihovny a před ukončením knihovny je provedena jejich hromadná destrukce. V implementaci využívají tento princip objekty reprezentující testovaný systém. Tento způsob není možné využívat pro cyklicky vytvářené proměnné jelikož nedochází k průběžnému uvolňování paměti za běhu.

4.1.5 Výstup testu

Výstupní data testu jsou tvořeny výstupním logem a v systému jsou reprezentována pomocí třídy TraceLogger. TraceLogger obaluje třídu TATraceLog, která uchovává jednotlivé řádky testu v objektu TATraceLine. Každý řádek logu zaznamenává jeden krok testu. Formát řádku logu je zachován z původního TASysTestu, aby bylo možné provést

načtení průběhu testu ze záznamu. V původním TASysTestu v jazyce C# byl formát logu definován pomocí technologie serializace. Serializace v jazyce C++ není defaultně podporována. Je sice možné nalézt knihovnu, např. Boost, která tuto funkcionalitu má. Lepším řešením je využít již použitou knihovnu TinyXML-2 [6] a proto je uložení záznamu TraceLogu naprogramováno explicitně. Formát výstupního logu je na obrázku 4.1

```
<Trace>
  <TraceLines>
    <TraceLine>
      <Time>316.0762733062669</Time>
      <TemplateInstance>0</TemplateInstance>
      <EdgeTaken>0</EdgeTaken>
      <SyncTemplateInstance>1</SyncTemplateInstance>
      <SyncEdgeTaken>0</SyncEdgeTaken>
      <VariableDump>
        <string>var id0_state_btn1: Integer, value = 0</string>
        <string>var id0_state_light1: Integer, value = 0</string>
        <string>var id0_timeout_light1: Integer, value = 0</string>
        <string>var id2_clk: Clock, value = 0</string>
      </VariableDump>
      <EventString> </EventString>
    </TraceLine>
  </TraceLines>
</Trace>
```

Obrázek 4.1. XML struktura výstupního logu testu. Zvýrazněné položky se mohou opakovat.

4.1.6 Statická třída StringUtil

Popis testovaného systému je reprezentován v textové podobě. Pro jeho zpracování je nutné použít pokročilé funkce nad řetězci. V C# jsou takové funkce samozřejmostí. V C++ je nutné si buď přilinkovat nějakou knihovnu nebo si funkce vytvořit vlastní. Byla zvolena možnost vlastní implementace požadované funkčnosti. Všechny implementované funkce jsou statické ve třídě StringUtil. Jedná se o následující metody:

- **string & trim(string & str)**

  Metoda provádí ostranění prázdných znaků na začátku a na konci řetězce *str*. Vstupní parametr je modifikován a návratou hodnotu tvoří reference na něj.

- **vector¡string¿ * tokenize(string & str, string delimiters, string & environmentNewLine)**

  Metoda provádí rozdělení řetězce *str* na jednotlivé tokeny. Oddělovač pro tokeny je definován parametrem *delimiters*. Parametrem *environmentNewLine* je prováděno nahrazení všech výskytů znaku $\backslash n$. Návratovou hodnotou je pointer na vector obsahující jednotlivé tokeny. Volající je povinnen provést destrukci vectoru.

- **string intToString(int value)**

  Metoda provádí převod čísla typu int na string.

- **int toInt(string & value, int & retVal)**

Metoda provádí převod string řětězce na typ int. Výsledek je v návratové hodnotě funkce a také v parametru *retVal*.

- **string & replace(string & str, string oldSubString, string newSubString)**.

  Metoda provádí nahrazení všech výskytů stringu *oldSubString* stringem *newSubString*. Návratovou hodnotou je reference na parametr *str*.

- **bool intToBool(int value)**

  Metoda provádí převod čísla typu int na typ bool.

4.1.7 Časová značka - třída HeartBeat

Při zaznamenávání kroku testu je nutné zaznamenat časovou značku. K tomuto účelu slouží statická metoda *double getTime()* třídy HeartBeat. Tato metoda pro získávání času využívá QueryPerformanceCounter [13]. QueryPerformanceCounter a jeho příbuzné metody jsou považovány za nejpřesnější zdroj času, který je možné v jazyce C++ získat. Třída Heartbeat v původní implementaci používá objekt StopWatch, který také využívá QueryPerformanceCounter.

4.1.8 Export funkcí z dll knihovny

Export funkcí z DLL [15] knihovny je možné jednak úpravou definice hlavičky každé funkce nebo pomocí souboru Definition file. Byl zvolen druhý způsob. Jedná se o přehlednější variantu. Soubor obsahuje název knihovny a jména exportovaných metod. Jiné metody, než ty které jsou uvedeny v tomto souboru není možné z dll volat. V projektu má tento soubor název *TASysTest_RealTime_Core.def* . Jeho název by se měl shodovat s názvem projektu.

Funkce knihovny jsou volány z prostřední NI LabView. LabView má odlišnou reprezentaci některých typů. Většina primitivních typů má shodnou reprezentaci. Největší rozdíl je v řetězcích. Řetězec v C++ je ukončován znakem \0. LabView tímto znakem řetězce neukončuje. Při předávání řetězců směrem do LabView nevzniká žádný problém. Znak \0 je interpretován správně. Problém vzniká opačným směrem. Proto při předávání řetězců směrem z LabView do C++ knihovny je nutné předat nejen samotný string, ale i jeho délku. Navíc LabView nepodporuje datový typ string, ale pouze pole řetězců.

4.1.8.1 Seznam exportovaných funkcí

Žádnou exportovanou funkci není možné korektně použít před voláním funkce TARTC_Initialize, která provede inicializaci knihovny.

Skupina funkcí reprezentující hlavní operace knihovny.

- **int TARTC_Initialize(char * inputFilePath, int length)**

  Tato metoda slouží pro inicializaci knihovny. Vstupní data metody tvoří parametr *inputFilePath*, což je cesta k souboru s testovaným modelem ve formě řetězce. Druhým parametrem je *length*, délka řetězce. Funkce vrací 0, pokud inicializace proběhla v pořádku, jiné číslo v případě chyby (např. pro neplatnou cestu k souboru vrací hodnotu 3).

- **int TARTC_Step()**

Tato metoda slouží pro provedení jednoho kroku testu. Předpokládá inicializovaný systém. Návratová hodnota je 1, pokud byl invariant splněn, v opačném případě je 0.

- **int TARTC_SaveTraceLog(char * outputFilePath, int length)**

  Provede uložení záznamů kroků testu do souboru, který je specifikovaný dvěma parmaetry - parametrem *outputFilePath*, kterým je specifikována cesta k souboru a parametrem *length*, který specifikuje délku řetězce *outputFilePath*. Návratová hodnota je 0, pokud se uložení podařilo, v opačném případě reportuje chybu.

- **int TARTC_Finalize()**

  Provede řádnou dealokaci knihovny. V případě úspěchu vrací hodnotu 1.

- **int TARTC_SaveRuntimeLog(char * logFilePath, int length)**

  Provede uložení záznamů runtime logu do souboru, který je specifikovaný dvěma parmaetry - parametrem *logFilePath*, kterým je specifikována cesta k souboru a parametrem *length*, který specifikuje délku řetězce *logFilePath*. Návratová hodnota je 0, pokud se uložení podařilo, v opačném případě reportuje chybu.

Skupina funkcí reprezentující přístup Custom Device k proměnným z Veristand Adaptoru.

- **void TARTC_CD_SetVariableValue_Name(char * varName, int varNameLength, double value)**

  Provede provede přiřazení hodnoty do proměnné podle zadadného jména.

- **double TARTC_CD_GetVariableValue_Name(char * varName, int varNameLength)**

  Vrátí hodnotu proměnné podle zadaného jména.

- **void TARTC_CD_SetVariableValue_Index(int varIndex, double value)**

  Provede provede přiřazení hodnoty do proměnné podle zadadného indexu.

- **double TARTC_CD_GetVariableValue_Index(int varIndex)**

  Vrátí hodnotu proměnné podle zadaného indexu.

- **int TARTC_GetNumOfVariables()**

  Vrátí celkový počet globálních proměnných v načteném modelu.

- **char * TARTC_GetVariableName(int index)**

  Vrátí jméno proměnné podle jejího indexu.

Skupina funkcí vracející příznaky souběhu metody Step.

- **int TARTC_Get_Step_Reading()**

Vrátí 1, pokud metoda Step čte ze společných proměnných VeristandAdaptoru, v opačném případě vrátí 0.

- **int TARTC_Get_Step_Writing()**

  Vrátí 1, pokud metoda Step zapisuje do společných proměnných VeristandAdaptoru, v opačném případě vrátí 0.

- **int TARTC_Get_Step_Running()**

  Vrátí 1, pokud metoda Step běží, jinak 0.

- **int TARTC_Get_Step_News()**

  Vrátí 1, pokud metoda Step zapsala do společných proměnných, jinak 0.

Skupina funkcí vracející a nastavující příznaky souběhu od Custom Device. Závislé na implementaci 3.strany. V knihovně nejsou tyto příznak nastavovány. (Např. z důvodu urychlení čtení a zapisování dat).

- **int TARTC_Get_CD_Reading()**

  Vrátí 1, pokud Custom Device čte hodnotu společných proměnných, jinak 0.

- **int TARTC_Set_CD_Reading(int value)**

  Nastaví proměnnou CD_Reading na požadovanou hodnotu. Měla by být nastavována na 1, pokud Custom Device čte proměnné z Veristand Adaptoru, jinak musí být 0. Vrátí nastavenou hodnotu.

- **int TARTC_Get_CD_Writing()**

  Vrátí 1, pokud Custom Device čte hodnotu společných proměnných, jinak 0.

- **int TARTC_Set_CD_Writing(int value)**

  Nastaví proměnnou CD_Writing na požadovanou hodnotu. Měla by být nastavována na 1, pokud Custom Device zapisuje proměnné do Veristand Adaptoru, jinak musí být 0. Vrátí nastavenou hodnotu.

- **int TARTC_Get_CD_News()**

  Vrátí hodnotu proměnné CD_News. Pokud Custom Device zapsal proměnné do VeristandAdaptoru, měla by být 1, v opačném případě 0.

- **int TARTC_Set_CD_News(int value)**

  Nastaví proměnnou CD_News na požadovanou hodnotu. Pokud Custom Device zapsalo proměnné do VeristandAdaptoru, měla by být nastavena na 1, v opačném případě na 0.

Funkce pro účely testování knihovny.

- **void TARTC_DummyMethod()**

  Prázdná metoda, nic nevykonává, nic nevrací. Slouží pro potřeby testování.

- **int TARTC_GetX()**

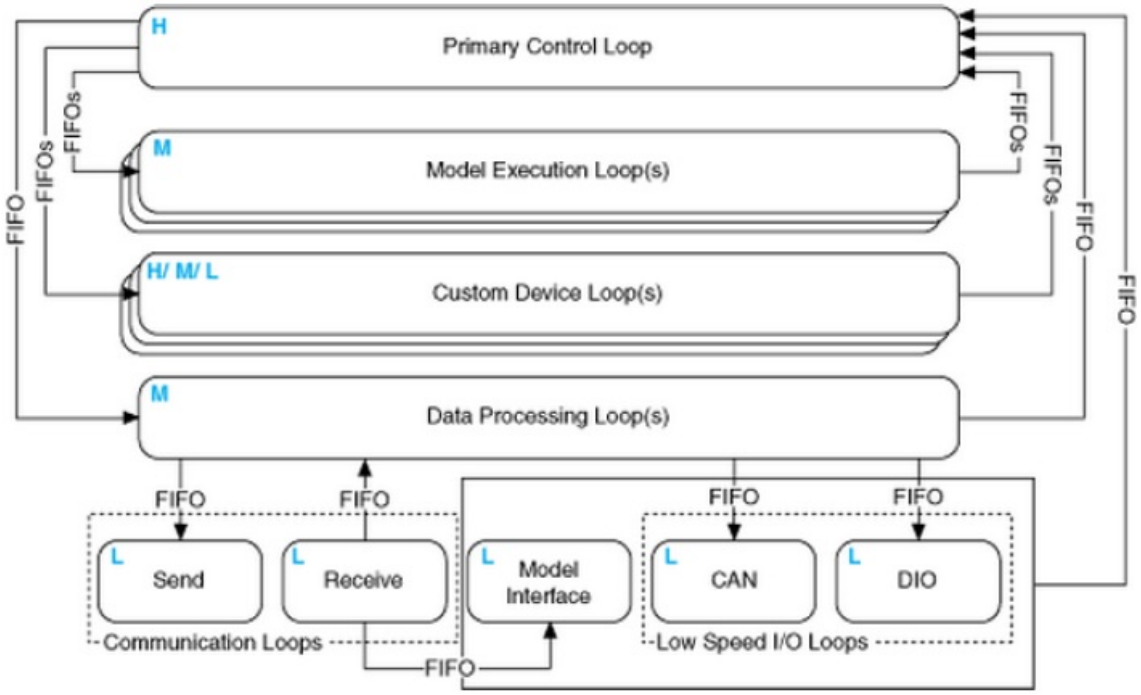
  Vrací hodnotu lokální statické proměnné X. Slouží pro potřeby testování.

4.2 Custom Device

Vývoj Custom Device probíhal s asistencí příručky [5].

Custom device je využito pro rozšíření funkcí NI VeriStandu o funkci výpočetního jádra TASysTestu.

Pro správnou implementaci je nutné znát NI Veristand Engine [11], který je na obrázku 4.2. NI Veristand pracuje v několika smyčkách. V každé iteraci přečte vstupy, provede patřičné operace a zapíše výstupy. Hlavní smyčkou je tzv. Primary Control Loop, která ostatní smyčky řídí[1]) a od níž je odvozeno časování systému. Ostatní smyčky pracují paralelně k PCL a s různou prioritou vykonávání. Mohou pracovat na stejné frekvenci jako PCL nebo i pomaleji než PCL. Komunikace mezi smyčkami je zajištěna pomocí vnitřních komunikačních FIFO kanálů. Custom Device může být spouštěno buď ve vlastní smyčce nebo přímo ve smyčce PCL.



Obrázek 4.2. Veristand Engine [11]

4.2.1 Návrh Custom Device [9]

Před startem implementace je nutné provést rozvržení základních parametrů:

- **Rozvržení kanálů - Channels(DBL)**

  Kanály jsou pouze datového typu 64-bitový double. Tento bod představuje rozvržení vstupních a výstupních kanálů v Custom Device. Vstupní kanály přenáší data ze zbytku NI Veristandu do Custom Device a výstupní kanály opačným směrem. V našem případě budou vstupní kanály Custom Device představovat vstupní data DLL knihovny a výstupní kanály budou představovat výstupní data DLL knihovny. Resp. Ze vstupních kanálů bude DLL data číst a do výstupních data zapisovat. Počet vstupních a výstupních kanálů bude proměnný v závislosti na testovaném

[1]) osnova vykonávání jednotlivých kroků PCL je zde [12]

systému. Proto je kladen požadavek na dynamické generování kanálů podle globálních parametrů testovaného systému. Dva kanály budou stálé napříč všemi modely - vstupní kanál pro možnost pozastavení testování s názvem *Step_Stop* a výstupní kanál *Step_Invariant_Satisfied*, který bude indikovat výsledek jedné iterace - resp. stav modelu.

- **Rozvržení konfiguračních parametrů - Properties (any data type)** Parametry jsou proměnné, jejichž hodnoty jsou nastavovány před startem projektu v RT targetu. Jejich nastavení je prováděno v System Exploreru NI Veristandu, v záložce Custom Device. Po kliknutí na konkrétní Custom Device je zobrazena tzv. Main Page, na které je možné nastavit některé parametry. V případě našeho Custom Device je nutné nastavit následující parametry: [1])
  - nastavení cest na host pc (operátorské): cesta k testovanému modelu, cesta pro zapsání výsledku logu, název souboru s logem,
  - nastavení parametrů pro rt target: IP adresy targetu, cílové složky na rt targetu, jména a hesla pro ftp přenos vstupního souboru do rt targetu,
  - nastavení parametrů testování: výběr testovací strategie.

- **Hierarchie Custom Device**

  Hierarchie Custom Device udává uspořádání vstupních a výstupních kanálů do skupin. Její správný návrh zjednodušuje obsluhu zařízení. Hierarchie může být buď tzv.plochá nebo strukturovaná. V našem případě byla zvolena strukturovaná hierachie - zvláští kategorie pro automaticky generované vstupy(*Model Inputs*), automaticky generované výstupy(*Model Outpus*). Kanály *Step_Stop* a *Step_Invariant_Satisfied* jsou v kořenovém umístění.

- **Extra Pages**

  Extra Pages jsou speciální vi, která jsou zobrazena při kliknutí na položku v Custom Device hierarchii. Pokud není Extra Page definována explicitně, použije se imlicitní verze. Pro naše účely nebylo nutné definovat žádnou Extra Page.

- **Typ Custom Device**

  Typ Custom Device je nutné vybrat podle účelu Custom Device. Existuje framework pro tři základní typy Custom Device - Asynchronní, Inline HW a Inline Model Interface. Rozdíl mezi nimi spočívá ve způsobu, jakým je PCL vykonává.

  Asynchronní je vykonáváno v paralelní smyčce vzhledem k PCL. Může běžet pomaleji, rychleji nebo stejně [2]) jako PCL. Tento typ Custom Device je možné synchronizovat k PCL, avšak determinističnost není stejně zaručena. Navíc má tu nevýhodu, že jeho typ frameworku nepodporuje záspis a čtení dat z dynamicky generovaných kanálů. Proto je pro naše účely zásadně nepoužitelné.

  Inline Custom Device je vykonáváno uvnitř PCL. Je zajištěna determinističnost vykonávání. Umožňuje číst a zapisovat dynamicky generované kanály. Je strukturováno jako stavový automat. Zde je rozdíl mezi Inline HW a Inline Model Custom Device - Smyčka Inline HW device je vykonává ve dvou fázích, viz. [12], zatímco v Inline Model Interface Custom Device je vykonávána v jedné fázi. Jak je již z názvu patrné, Inline HW je primárně určeno pro komunikaci s hardwarem. Vzhledem k

[1]) Viz. 5.1.
[2]) tzv. pseudosynchronně

osnově vykonávání smyčky PCL - [12] Inline HW přečte data z hw, poskytne je na začátku iterace PCL, proběhnou ostatní úkony a poté jsou data zapsána do fyzického hw. V našem případě by při použití tohoto typu vznikaly komunikační prodlevy. Inline Model čte data ze systému v době, kdy jsou již získána ze všech ostatních částí systému, provede svůj kód a ihned poskytne výsledky zbytku systému. Z těchto důvodů bylo zvoleno Inline Model Custom Device.

Srovnání jednotlivých Custom Device je na obrázku 4.3.

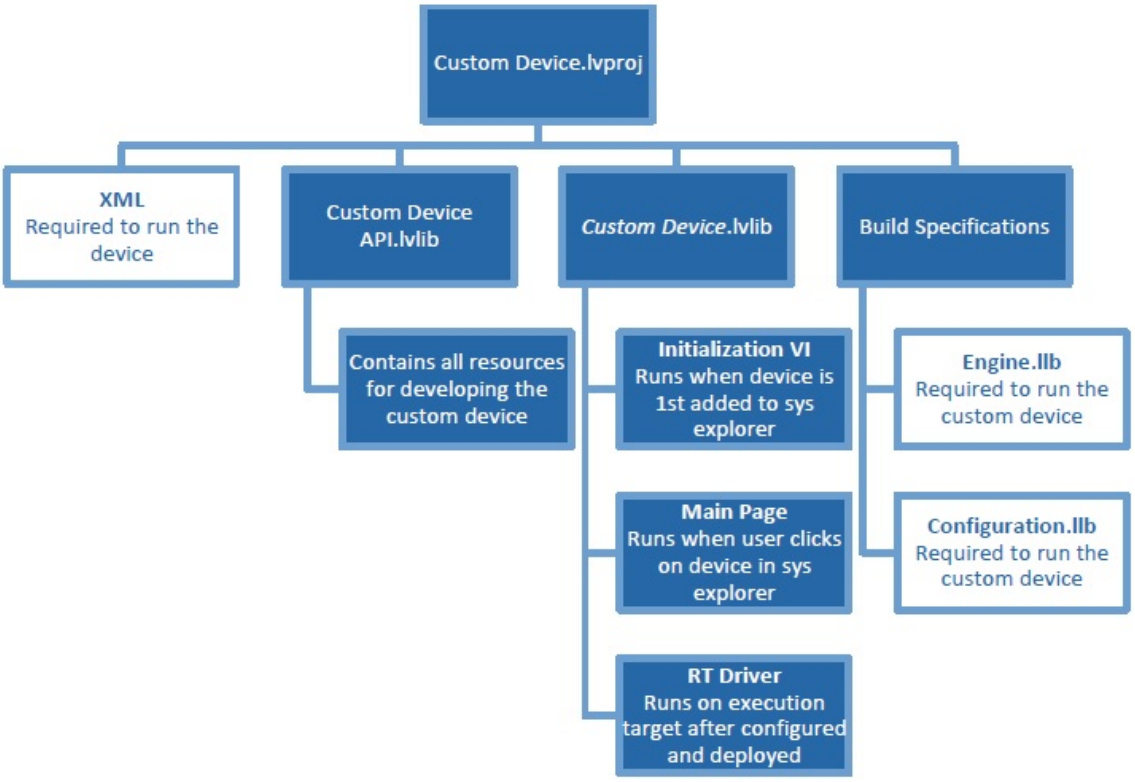
Table of Custom Device Frameworks

Device Type	Basic Architecture	Framework Data Interface	Timing	Pros	Caveats	Use Cases
Asynchronous	Single Loop	Input and Output FIFO	Synchronized w/ PCL Decimation of PCL rate (FIFOs are read ever N'th iteration of PCL) Any user defined rate	Unlikely to adversely affect timing of other components in the system May run faster, slower, or decimation of PCL	1-cycle latency to get data to/from the device due to RT FIFOs	Shared resources, background processes, non-deterministic hardware/ protocols, system health monitoring, logging, offline analysis
Inline Hardware Interface	State machine Two steady-state cases	Channel references	In-line with the PCL Decimation of the PCL (device executes every N'th iteration of PCL, does not have N-times as long to finish)	Presents data to engine before other components execute Receive data from engine after other components have executed	Can adversely affect the timing of the PCL	Most hardware, deterministic operations, two-phase operations such as stimulus-response
Inline Model Interface	State machine One steady-state case	Channel references	In-line with the PCL Decimation of the PCL (device executes every N'th iteration of PCL, does not have N-times as long to finish)	Send data to engine with low latency	Can adversely affect the timing of the PCL	Low-latency calculations such as PID, interpolation, etc.

Obrázek 4.3. Srovnání jednotlivých druhů Custom Device - není přeloženo. Převzato z [5].

4.2.2 Implementace

Implementace Custom Device byla prováděna pomocí nástroje Custom Device Template Tool. Tento nástroj vygeneruje podle nastaveného typu Custom Device potřebný framework. Vygenerovaný framework byl dále upraven. Struktura vygenerovaného frameworku je na obrázku 4.4.



Obrázek 4.4. Custom Device Framework [8]

Volání DLL knihovny z Custom Device je prováděno pomocí tzv. Call Library Function Node.

Popis vybraných položek frameworku

- **Initialization VI**

  Toto VI běží na host pc a je spuštěno pokaždé, když je vytvořena nová instance Custom Device. V TARTC Custom Device je využito pro inicializaci globálních proměnných a pro přidání potřebných závislostí.

- **Main Page VI**

  Toto VI běží na host pc a je spouštěno pokaždé, když se klikne na název instance Custom Device. V TARTC Custom Device je využito pro jeho konfiguraci. Jeho funkce v TARTC Custom Device je popsána v 5.2. Mimo zadávání konfiguračních údajů implementuje funkce pro dynamické přidávání kanálů podle testovaného modelu a také FTP přenos souborů mezi Host PC a RT Targetem.

  Dynamické generování kanálů V TARTC projektu k tomuto účelu slouží VI Add Channels z knihovny TARTC Custom Device.lvlib. Dynamické generování kanálů je prováděno následovně. Již v této chvíli je využívána DLL knihovna TASysTest Real-Time Core. V tomto případě však běží ještě na host pc. Knihovna musí být nejprve inicializována. Po správném inicializaci je načten a naparsován model systému. Pomocí funkcí, které knihovna poskytuje je zjištěno kolik je v modelu globálních proměnných a poté jsou v cyklu zjišťovány jejich jména, pod kterými jsou vytvářeny nové kanály v Custom Device. Kanály jsou dynamicky vytvářeny pomocí funkce Add

Channel.vi. Algoritmus je primitivní a je velmi jednoduché pochopit jeho princip studiem zdrojového vi.

- **RT Driver**

  Toto VI běží v RT targetu. Obsahuje stavový automat jehož stavy jsou postupně spouštěny v PCL. V případě Inline Model obsahuje **4 stavy**:

  - **Initialize** Tento stav je spuštěn jednou při deploynutí VeriStand projektu do targetu. Je spuštěn před stavem Start.

    V TARTC CD je v něm provedeno získání referencí na kanály.
  - **Start** Stejně jako stav Initialize je tento stav spuštěn jednou při deploynutí VeriStand projektu do targetu. Je spuštěn po stavu Initialize.

    Zde je provedena inicializace DLL TASysTest Real-Time Core.
  - **Execute Model** Tento stav je spouštěn v PCL smyčce.

    Zde je vykonáván krok testu. Nejprve je provedeno zapsání hodnot proměnných z kanálů do knihovny, poté je vykonán krok testu a nakonec je proveden zápis proměnných z knihovny do kanálů.
  - **Close** Tento stav je spuštěn při undeploynutí projektu z targetu

    V tomto stavu se provádí uložení TraceLogu a RuntimeLogu do souborů a je zavolána finalize metoda knihovny.

Je důležité poznamenat - Při volání DLL z LabVIEW pomocí Call Library Function Node nedokáže LabVIEW přerušit vykonávání DLL. Proto pokud by bylo vykonávání příliš dlouhé, v případě Inline Custom Device by docházelo k ovlivňování real-time časování PCL smyčky. Vzhledem k tomu je třeba volit frekvenci PCL smyčky.

Kapitola 5
TARTC Custom Device

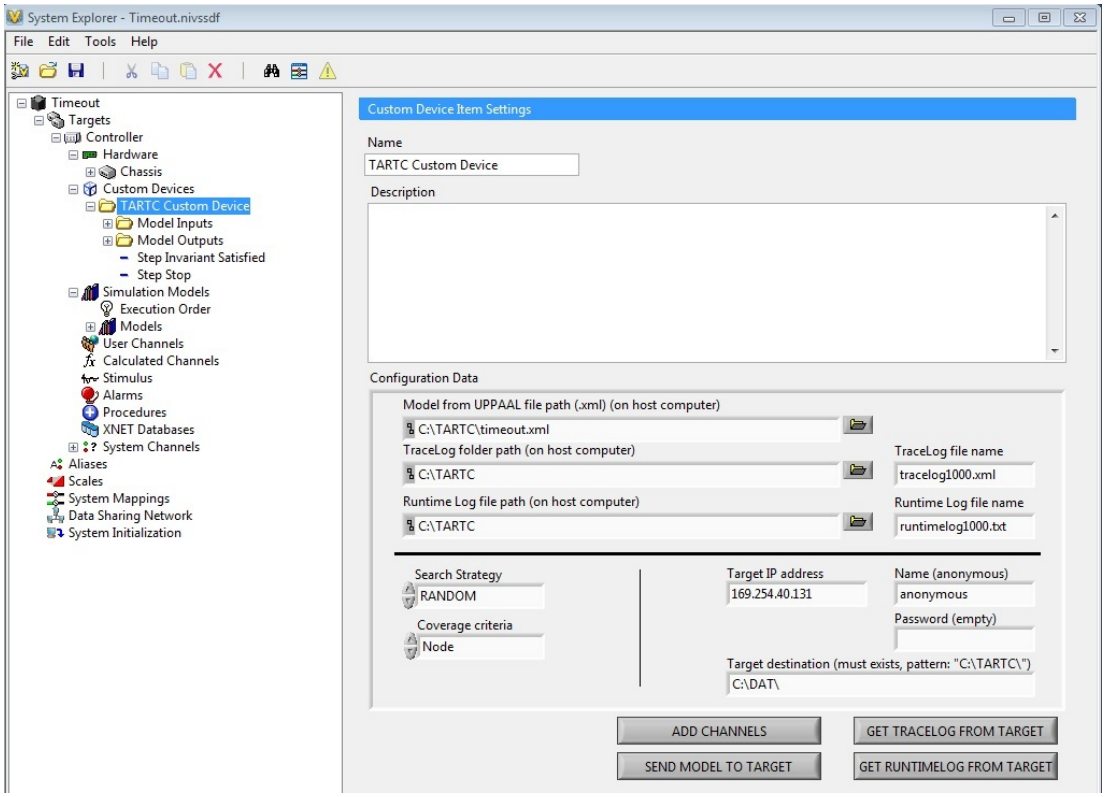
5.1 Distribuce Custom Device

Custom device je distribuováno ve dvou formách - ve formě zkopilovaného Custom Device připravené pro použití v NI VeriStand a ve formě distribuce zdrojových kódu. Na přiloženém CD se nachází ve složce NI LabView/TARTC Custom Device, viz.A.1. Ve složce source jsou zdrojové soubory, ve složce Build je zkompilovaná knihovna. Kompilace je prováděna verzí NI Veristand 2014. Pro jinou verzi je nutné si Custom Device zkompilovat [8]. Pro zobrazní Custom Device v NI Veristandu je nutné nakopírovat jeho zkompilovanou verzi (celou složku TARTC Custom Device, viz.A.1) do tzv. *Common Data Dir*. Pro Windows 7 je defaultní cesta C:\Users\Public\Documents\National Instruments\NI VeriStand 2014 pro jinou verzi VeriStandu jiné jméno poslední složky. Pokud je zkopilované zařízení v této destinaci, je možné jej přidat do projektu v NI Veristand.

5.2 Popis obsluhy

Přidání Custom Device do projektu v NI Veristand se provádí v System Exploreru v souboru System Definiton File v sekci Controller/Custom Devices. Po přidání Custom Device do NI Veristand je zobrazena konfigurační stránka. Konfigurační stránka TARTC Custom Device je zobrazena na obrázku 5.1. Pro každý testovaný model je nejprve nutné vygenerovat komunikační kanály. Pokud v hierarchii Custom Device/TARTC Custom Device existují složky *Model outputs a Model Inputs* je nutné je neprve odstranit. Po jejich odstranění je možné přejít ke generování kanálů. V poli *Model from UPPAAL file path* se vybere testovaný model. Po jeho vybrání se automatické generování kanálů provede stisknutím tlačítka *ADD CHANNELS*. Pokud je vybrán špatný soubor nebo pokud není cesta platná, nic se nestane. Kanály se jenom nevygenerují. Po úspěšném vygenerování se zobrazí v hierarchii Custom Device. Dále je nutné přenést testovaný model do rt targetu. Je nutné nastavit správnou IP adresu (pole *Target IP address* a správnou destinaci na cílové platformě (pole *Target destination*. Pozor, cesta musí existovat, jinak dojde k ohlášení chyby a soubor se nepřenese. Cesta musí být zadána podle vzoru nad políčkem. Resp. písmeno jednotky musí být celké, lomítka zpětná a poslední lomítko nesmí chybět. Pokud není na rt targetu explicitně přenastaveno přístupové jméno a heslo, nesmí se pole *Name* a pole *Password* měnit. Po tomto nastavení je možné přenést soubor na rt target tlačítkem *MODEL TO TARGET*. Pro běh testu je dále nutné nastavit vyhledávací strategii. Defaultní vyhledávací strategie je náhodá. Ostatní jsou zatím experimentální. Nakonec je nutné nastavit jména souborů výstupních logů testu, pod kterými budou výsledky v targetu uloženy. Budou uloženy do nastavené destinace. Po skončení deploynutí a undeploynutí projektu na rt targetu jsou logy dostupné v nastavené destinaci. Pro jejich přenos do host pc je možné využít tlačítka *GET TRACELOG FROM TARGET a GET RUNTIMELOG FROM TARGET*.

Při správném nastavení IP adresy, přihlašovacího jména, hesla, destinace a názvu souborů budou soubory po stisku tlačítka přeneseny do umístění, které se nastavuje v poli *TraceLog folder path* a *Runtime Log file path*. Při špatných nastaveních vyskočí chybové hlášení nic se nepřenese. Mapování kanálů do zbytku systému se provádí klasickým způsobem v sekci System Configuration Mapping, viz. 6.2.



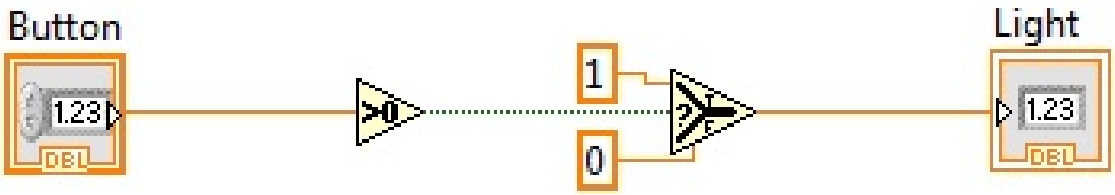
Obrázek 5.1. TARTC Custom Device konfigurační stránka.

Kapitola 6
Demostrace funkčnosti na NI PXIe-8135

V této kapitole je proveden popis demostrace funčnosti v prostředí NI Real-Time Phar Lap ETS v targetu NI PXIe-8135.

6.1 Demonstrace funkčnosti

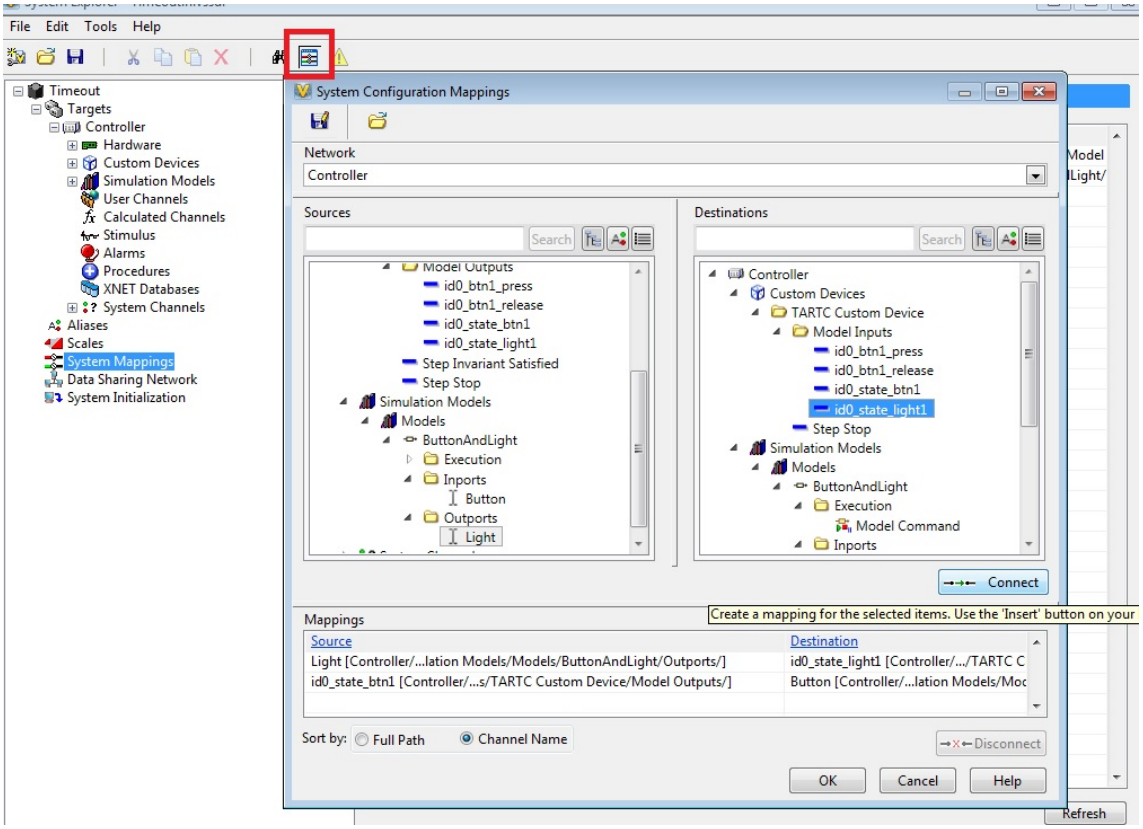
V této kapitole je proveden popis ověření funkčnosti implementovaného nástroje. Ověření je provedeno na modelu systému tlačítka a vypínače. Popis tohoto modelu je uveden zde 2.2.3. Jedná se o model vytvořený v UPPAAL, který ověřuje rozsvícení světla do uplynutí nějakého timeoutu. Pro testování nebyl využit reálný hardware, ale jednoduchý simulační model vytvořený v NI LabView a zkompilovaný pro běh v NI VeriStand. Jméno modelu, který simuloval chování HW je ButtonAndLight. Výsledky testy jsou uloženy v TraceLoguB. Pro průběh testu je vytvořeno patřičné Workspace??.



Obrázek 6.1. Model světla a vypínače nahrazující reálný hardware použitý pro testování funčnosti implementace.

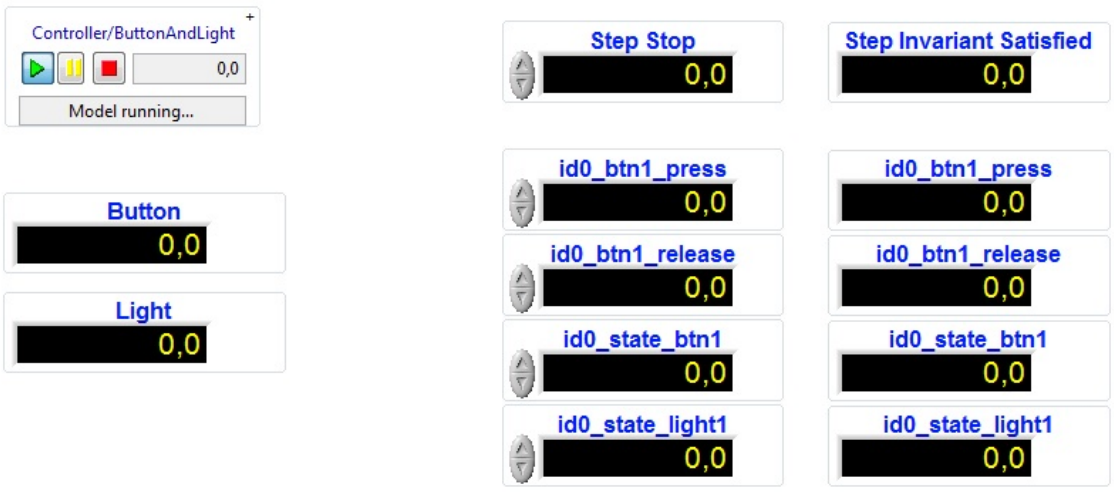
Postup vytvoření projektu pro demostraci funčnosti

TARTC Custom Device i simulační model ButtonAndLight musejí být přidány do jednoho projektu v NI Veristandu. Custom Device ve své sekci a simulační model v sekci Simulation Models. Před spuštěním testu je nutné provést mapování kanálů v NI Veristandu. Mapování se provádí v System Exploreru v sekci System Configuration Mappings.(Tlačítko červené vyznačené na obrázku6.2. Mapováním se propojí patřičné vstupy a výstupy testovaného a simulačního modelu. Konrétně dojde k propojení výstupu *Light* simulačního modelu se vstupem *id0_state_light1* a propojení výstupu testovaného modelu *ido_state_btn1* se vstupem*Button* simulačního modelu. Po provedení správné mapování je jetě nutné nastavit IP adresu targetu a frekvenci PCL smyčky na požadovanou hodnotu. U simulačního modelu je nutné nastavit decimaci na hodnotu 6. Pro zobrazení hodnot kanálů během testuje dobré vytvořit patřičnou workspace. Pokud jsou provedené všechny kroky, může se projekt deploynout na rt target.



Obrázek 6.2. Ukázka propojování kanálů mezi Custom Device a simulačním modelem.

Po deploynutí projektu začne probíhat testování v rt targetu. Při undeploynutí dojte k zapsání souborů do nastavené lokace na rt targetu. Výsledky je možné přenést z RT targetu pomocí tlačítka Custom Device v System Exploreru nebo je možné využít alternativní možnost, například NI MAX. Výsledky logu testu jsou v příloze B.



Obrázek 6.3. Navržená workspace pro testování funčnosti implementace.

Kapitola 7
Závěr

7.1 Zhodnocení výsledků

Cílem práce bylo provést modifikaci nástroje TASysTest pro běh v prostředí real-time s následnou demonstrací funkčnosti provedené implementace. Na začátku práce byly navrženy úpravy architektury se snahou dodržet zadané požadavky. Nejprve se provedlo seznámení s původním nástrojem TASysTest. Poté byly zvoleny vhodné nástroje a navrženy úpravy architektury pro běh v prostředí NI Real-Time PharLap ETS. Portace byla provedena implementací jádra v jazyce C++ formou DLL knihovny, která byla následně integrována do NI VeriStand prostředí prostřednictvím Custom Device.

Bylo provedeno odzkoušení funkčnosti provedené implementace v prostředí NI Real-Time PharLap ETS nainstalovaném na desktopovém PC. V závěru práce byla funkčnost implementace odzkoušena v NI PXIe-8135 targetu. Implementace byla funkční, bez zjevných problémů běhu. Odzkoušení bylo provedeno pomocí simulačního modelu. Testování bylo prováděno na frekvenci 100Hz. V NI PXIe-8135 časové intervaly jednotlivých kroků testu dosahovaly přednosti desetiny milisekundy. Z časových důvodů nebyl proveden test real-time chování implementace.

Portací výkonného jádra TASysTestu přímo do prostředí NI Veristand přináší odstranění komunikačních latencí, které vznikaly prostřednictvím .NET API. Dalším přínosem práce je ukázka funkčnosti [16] DLL knihovny vyvíjené v prostředí MS Visual Studio 2010 v prostředí NI Real-Time PharLap ETS. Přínosem je i nalezení fungujícího TinyXML-2 [6] open source C++ XML parseru v real-time prostředí. Hlavním přínosem mělo být dosažení deterministického chování testování. Real-time chování implementovaného systému nebylo z časových důvodů ověřeno a proto není možné určit, zda byl tento cíl splněn.

7.2 Další práce

- **Otestování real-time chování nástroje**

  Bylo by dobré provést ověření časového chování implementovaného nástroje, které se v této práci nepodařilo stihnout.

- **Automatizování přenosu souborů do az RT targetu**

  Před začátkem testování musí operátor provést přenos testovaného modelu do RT targetu. Při tom musí vědět, jaká cesta na RT targetu existuje, jinak neproběhne přenos v pořádku. Vhodným rozšířením by bylo automatické vytváření zadané cesty na rt targetu. Při zahájení testu by mohl být automaticky ze zadaných cest proveden přenos testovaného modelu na RT target a po skončení testu by se výsledné logy mohly přenést do pc.

- **Real-time vizualizace průběhu testování v uživatelském rozhraní původního TASysTestu**

  Při průběhu testu na RT targetu nejsou k dispozici žádné aktuální informace. Průběh testu by bylo hezké vizualizovat pomocí TASysTestu. Samozřejmě v závislosti na rychlosti průběhu testování. Technologií TCP/IP by mělo být možné přenášet data z a do RT targetu při jeho běhu. V Custom Device je k tomuto účelu používáno Inline Custom Device se pseudoasynchronní smyčkou spouštěnou programově z Inline.

- **Automatizace spouštění z původního nástroje TASysTest**

  Původním požadavkem na výsledný nástroj bylo automatizované ovládání spouštění real-time verze výpočetního jádra přímo z nástroje TASysTest. Existuje .NET API, které by to mělo umožňovat.

Literatura

[1] Tomáší Grus. *Implementace softwarového nástroje pro generování integračních testů*. 2014.

[2] J. Zander, I. Schieferdecker a P.J. Mosterman. *Model-Based Testing for Embedded Systems*. Taylor & Francis, 2011.

[3] Jon CONWAY a Steve WATTS. *A software engineering approach to LabVIEW*. Upper Saddle River, NJ: Prentice Hall, Professional Technical Reference, c2003, xiii, 221 p., 2011. ISBN 01-300-9365-3.

[4] Miroslav Virius. *Programovánív C++*. Praha: ČVUT: 1988. ISBN ISBN 978-80-01-04371-4.

[5] National Instruments Corporation. *NI VeriStand 2010 Custom Device Developers Guide (Beta)*.
http://download.ni.com/pub/devzone/tut/niveristand_cd_dev_guide.pdf.

[6] *TinyXML-2*.
http://www.grinninglizard.com/tinyxml2docs/.

[7] *UPPAAL*.
http://www.uppaal.org/.

[8] *Building Custom Devices for NI VeriStand*.
http://www.ni.com/tutorial/9348/en/.

[9] *Planning a Custom Device*.
http://zone.ni.com/reference/en-XX/help/372846J-01/veristandmerge/planning_cds/.

[10] *What Is VeriStand?*
http://www.ni.com/veristand/whatis/.

[11] *Understanding the VeriStand Engine*.
http://zone.ni.com/reference/en-XX/help/372846G-01/veristand/understanding_vs_engine/.

[12] *List of PCL Execution Steps*.
http://zone.ni.com/reference/en-XX/help/372846G-01/veristand/pcl_execution.

[13] *Acquiring high-resolution time stamps*.
https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408.aspx.

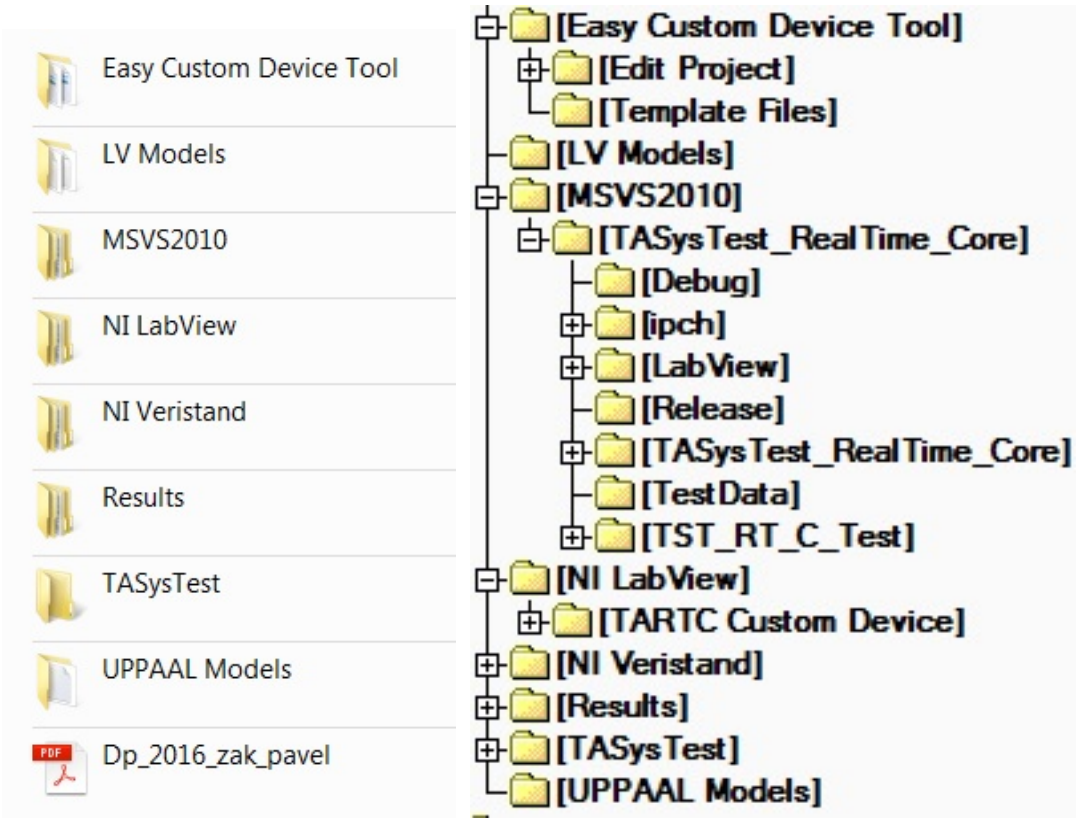
[14] *Building a DLL with Visual C++*.
http://www.ni.com/white-paper/3056/en/.

[15] *Exporting from a DLL Using DEF Files*.
https://msdn.microsoft.com/en-us/library/d91k01sh.aspx.

[16] *How Can I Verify That My DLL Is Executable in LabVIEW Real-Time on NI PharLap ETS?*
http://digital.ni.com/public.nsf/allkb/0BF52E6FAC0BF9C286256EDB00015230.

Příloha A
Obsah CD

A.1 Obsah kořenového adresáře *root*



Obrázek A.1. Obrázek zobrazuje obsah kořenového adrešáře CD. Vlevo jsou vidět složky i soubory. Vpravo jsou rozbalené složky z úrovně root.

Příloha B
Výstup testování - záznam TraceLog

B.1 Ověření funkce systému - Měření odezvy události

Ověření funkce probíhá na operčním systému OS NI Real-Time Phar Lap ETS. Frekvence PCL smyčky NI Veristandu je nastavena na hodnotu 100Hz, tzn. 10ms. Zaznamenaný čas v logu je v jednotkách ms.

B.1.1 Na běžném desktopovém PC

konfigurace:

- Intel Pentium 4 2.4GHz,
- 1GB RAM,
- HDD Seagate Barracude ATA, Model ST360021A 60GB,
- síťová karta Intel PRO 1000GT Desktop Adapter.

```
        <TraceLine>
            <Time>319.73284200000001</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 3</string>
                <string>var id2_light_clk: Clock, value = 3</string>
            </VariableDump>
            <EventString>btn[0] took the edge from id0 () to id0 ()
            </EventString>
        </TraceLine>
        <TraceLine>
            <Time>329.72097500000001</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 4</string>
                <string>var id2_light_clk: Clock, value = 4</string>
            </VariableDump>
```

```
            <EventString>btn[0] took the edge from id0 () to id0 ()
            </EventString>
        </TraceLine>
        <TraceLine>
            <Time>339.72889800000002</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 5</string>
                <string>var id2_light_clk: Clock, value = 5</string>
            </VariableDump>
            <EventString>btn[0] took the edge from id0 () to id0 ()
            </EventString>
        </TraceLine>
        <TraceLine>
            <Time>349.73385300000001</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>1</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 6</string>
                <string>var id2_light_clk: Clock, value = 6</string>
            </VariableDump>
            <EventString>btn[0] took the edge from id0 () to
                                BTN_OFF</EventString>
        </TraceLine>
        <TraceLine>
            <Time>359.90635900000001</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>1</SyncTemplateInstance>
            <SyncEdgeTaken>0</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 1</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 0</string>
                <string>var id2_light_clk: Clock, value = 0</string>
            </VariableDump>
            <EventString>btn[0] took the edge from BTN_OFF
                                to BTN_ON triggering the sync
                                'id0_btn1_press?' of light[1] taking
                                the edge from START_NODE
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>369.76193799999999</Time>
```

```
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 1</string>
                <string>var id1_btn_clk: Clock, value = 1</string>
                <string>var id2_light_clk: Clock, value = 1</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>379.77888100000001</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 2</string>
                <string>var id2_light_clk: Clock, value = 2</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>389.75483700000001</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 3</string>
                <string>var id2_light_clk: Clock, value = 3</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>399.752861</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 4</string>
```

```
                <string>var id2_light_clk: Clock, value = 4</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>409.81269600000002</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>1</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 5</string>
                <string>var id2_light_clk: Clock, value = 5</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                        to LIGHT_ON</EventString>
        </TraceLine>
```

B.1.2 Ověření funkce pomocí NI PCIe-8135 targetu

```
<TraceLines>
    <TraceLine>
        <Time>6.0000000000000002e-006</Time>
        <TemplateInstance>-1</TemplateInstance>
        <EdgeTaken>-1</EdgeTaken>
        <SyncTemplateInstance>-1</SyncTemplateInstance>
        <SyncEdgeTaken>-1</SyncEdgeTaken>
        <VariableDump>
            <string>var id0_state_btn1: Integer, value = 0</string>
            <string>var id0_state_light1: Integer, value = 0</string>
            <string>var id1_btn_clk: Clock, value = 0</string>
            <string>var id2_light_clk: Clock, value = 0</string>
        </VariableDump>
        <EventString>Zero step</EventString>
    </TraceLine>
    <TraceLine>
        <Time>413.821932</Time>
        <TemplateInstance>0</TemplateInstance>
        <EdgeTaken>0</EdgeTaken>
        <SyncTemplateInstance>-1</SyncTemplateInstance>
        <SyncEdgeTaken>-1</SyncEdgeTaken>
        <VariableDump>
            <string>var id0_state_btn1: Integer, value = 0</string>
            <string>var id0_state_light1: Integer, value = 0</string>
            <string>var id1_btn_clk: Clock, value = 0</string>
            <string>var id2_light_clk: Clock, value = 0</string>
        </VariableDump>
        <EventString>btn[0] took the edge from id0 () to id0 ()
        </EventString>
    </TraceLine>
```

```
        <TraceLine>
            <Time>421.15684700000003</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 1</string>
                <string>var id2_light_clk: Clock, value = 1</string>
            </VariableDump>
            <EventString>btn[0] took the edge from id0 () to id0 ()
           </EventString>
        </TraceLine>
        <TraceLine>
            <Time>431.14780500000001</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>1</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 2</string>
                <string>var id2_light_clk: Clock, value = 2</string>
            </VariableDump>
            <EventString>btn[0] took the edge from id0 ()
                                to BTN_OFF</EventString>
        </TraceLine>
        <TraceLine>
            <Time>441.199794</Time>
            <TemplateInstance>0</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>1</SyncTemplateInstance>
            <SyncEdgeTaken>0</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 1</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 0</string>
                <string>var id2_light_clk: Clock, value = 0</string>
            </VariableDump>
            <EventString>btn[0] took the edge from BTN_OFF
                                to BTN_ON triggering the sync
                                'id0_btn1_press?' of light[1]
                                taking the edge from START_NODE
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>451.15974599999998</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
```

```
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 1</string>
                <string>var id1_btn_clk: Clock, value = 1</string>
                <string>var id2_light_clk: Clock, value = 1</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>461.15797799999996</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 2</string>
                <string>var id2_light_clk: Clock, value = 2</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>471.16586499999994</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 3</string>
                <string>var id2_light_clk: Clock, value = 3</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                                to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>481.15838399999996</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>0</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 4</string>
                <string>var id2_light_clk: Clock, value = 4</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
```

```
                            to SWITCHING_ON</EventString>
        </TraceLine>
        <TraceLine>
            <Time>491.17284599999994</Time>
            <TemplateInstance>1</TemplateInstance>
            <EdgeTaken>1</EdgeTaken>
            <SyncTemplateInstance>-1</SyncTemplateInstance>
            <SyncEdgeTaken>-1</SyncEdgeTaken>
            <VariableDump>
                <string>var id0_state_btn1: Integer, value = 0</string>
                <string>var id0_state_light1: Integer, value = 0</string>
                <string>var id1_btn_clk: Clock, value = 5</string>
                <string>var id2_light_clk: Clock, value = 5</string>
            </VariableDump>
            <EventString>light[1] took the edge from SWITCHING_ON
                            to LIGHT_ON</EventString>
        </TraceLine>
```

Příloha C
Zkratky a symboly

C.1 Zkratky

TASysTest	Timed Automata System Tester. Testovací prostředí pro vytváření a provádení integračních testů.
TST	TASysTest.
UPPAAL	Uppaal je integrované testovací prostředí pro modelování, validaci a verifikaci systémů reálného času modelovaných jako síť časovaných automatů rozšířených o datové typy.
NI	Společnost National Instruments.
PXI	PCI Extensions for Instrumentation, robustní PC platforma pro měření a automatizování systémů.
NI Veristand	Softwarové prostředí pro konfiguraci testovací aplikací v reálném čase od společnosti National Instruments.
NI LabView	Softwarové prostředí založené na grafickém programovacím jazyce G, od společnosti National Instruments.
.NET	Platforma založená na .NET Frameworku od společnosti Microsoft.
API	Application Programming Interface.
HW	Hardware.
OS	Operating system. Operační Systém.
XML	Extensible Markup Language. Přenositelný, platformě nezávislý formát dat.
HIL	Hardware-In-Loop, způsob testování systémů.
MS	Microsoft
DLL	Dynamic Link Library - dynamicky linkovaná knihovna, způsob znovupoužití stejného kódu